Specification and Inference of Trace Refinement Relations

TIMOS ANTONOPOULOS, Yale University, USA
ERIC KOSKINEN, Stevens Institute of Technology, USA
TON CHANH LE, Stevens Institute of Technology, USA

The modern software engineering process is evolutionary, with commits/patches begetting new versions of code, progressing steadily toward improved systems. In recent years, program analysis and verification tools have exploited version-based reasoning, where new code can be seen in terms of how it has changed from the previous version. When considering program versions, refinement seems a natural fit and, in recent decades, researchers have weakened classical notions of concrete refinement and program equivalence to capture similarities as well as differences between programs. For example, Benton, Yang and others have worked on state-based *refinement relations*.

In this paper, we explore a form of weak refinement based on *trace* relations rather than state relations. The idea begins by partitioning traces of a program C_1 into trace classes, each identified via a *restriction* r_1. For each class, we specify similar behavior in the other program C_2 via a separate restriction r_2 on C_2. Still, these two trace classes may not yet be equivalent so we further permit a weakening via a binary relation $\mathcal{A}$ on traces, that allows one to, for instance disregard unimportant events, relate analogous atomic events, etc.

We address several challenges that arise. First, we explore one way to specify trace refinement relations by instantiating the framework to Kleene Algebra with Tests (KAT) due to Kozen. We use KAT intersection for restriction, KAT hypotheses for $\mathcal{A}$, KAT inclusion for refinement, and have proved compositionality. Next, we present an algorithm for automatically synthesizing refinement relations, based on a mixture of semantic program abstraction, KAT inclusion, a custom edit-distance algorithm on counterexamples, and case-analysis on nondeterministic branching. We have proved our algorithm to be sound. Finally, we implemented our algorithm as a tool called Knotical, on top of Interproc and Symkat. We demonstrate promising first steps in synthesizing trace refinement relations across a hand-crafted collection of 37 benchmarks that include changing fragments of array programs, models of systems code, and examples inspired by the `thttpd` and Merecat web servers.

CCS Concepts: • **Theory of computation** → **Modal and temporal logics**; **Verification by model checking**; **Program verification**; *Abstract machines.*

Additional Key Words and Phrases: Program refinement, trace refinement, Kleene Algebra with Tests

ACM Reference Format:
Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. 2019. Specification and Inference of Trace Refinement Relations. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 178 (October 2019), 30 pages. https://doi.org/10.1145/3360604

1 INTRODUCTION

Modern software changes at a rapid pace. Software engineering practices, such as Agile, advocate an evolutionary software development process, where ongoing source code edits lead, slowly but surely, toward an improved system. Meanwhile, as these software systems grow, fragments of code

Authors' addresses: Timos Antonopoulos, Yale University, New Haven, CT, USA; Eric Koskinen, Stevens Institute of Technology, Hoboken, NJ, USA; Ton Chanh Le, Stevens Institute of Technology, Hoboken, NJ, USA.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2019 Copyright held by the owner/author(s).
2475-1421/2019/10-ART178
https://doi.org/10.1145/3360604

are reused in increasingly many different contexts. To complicate matters, these contexts themselves may be changing, and code written under some assumptions today may be used under different ones tomorrow. With so many moving parts, researchers and practitioners have found compositional reasoning across versions (*e.g.* [Logozzo et al. 2014; O'Hearn 2018]) to be indispensable.

Changes can be exploited for good purposes: they offer a sort of informal specification, where programmers often view their new code in terms of how it has deviated from the existing code (*i.e.* a commit or patch), including the removal of bugs, addition of new features, performance improvements, etc. With compositional theories and tools, one can reuse previous analysis results for unchanged code, and combine them with new analyses of only the changing code fragment.

It is therefore a natural question to ask: how does a given program C_1 compare to C_2, a modified version of C_1? If one is merely interested in knowing whether they are strictly equivalent (or whether, for example, C_1 is contained within C_2), such a notion is commonly referred to as *program refinement* and pertains to compiler correctness, translation validation [Pnueli et al. 1998], and the refinement calculus of Morgan [1994]. Intuitively, C_2 concretely refines C_1 provided that, when executed from the same initial state, they both reach the same final state. Researchers have developed algorithms and tools (*e.g.* [Lahiri et al. 2012, 2013; Wood et al. 2017]) to check whether, say, two versions of a function return the same results. Similarly, bisimulation provides *equivalence* between how programs behave over time, perhaps accounting for different implementations.

Concrete refinement and bisimulation are often not focused on how the programs differ, but simply whether or not they are equivalent. Some have since worked toward weakening program equivalence. For instance, the works of Benton [2004] and Yang [2007] allowed one to define equivalence relations over the state space, to express that two programs reach the same output equivalence relation when executed from states in a particular input equivalence relation. Such equivalences allow one to describe what differences over the states one does or does not care about, for example, focusing on important variables or ignoring scratch data. This strategy is compositional because one can correlate the output relation of one code fragment with the input relation of the next. Extensions were later explored by others [Gyori et al. 2017; Unno et al. 2017].

1.1 Toward Trace Refinement Relations

While the above works have focused on *state relations*, we instead build toward a refinement theory of *trace relations*. Our motivating goal is to be able to express similarities/differences between two different programs in terms of their behaviors over time. Examples include whether two programs send/receive messages in the same order, follow the same allocation/release orders, have features added/removed, or have similar certain I/O patterns. Naturally, we take a reactive view of programs, treating their execution in terms of *events*, which can be suitably defined in terms of statements, function calls, I/O, etc.

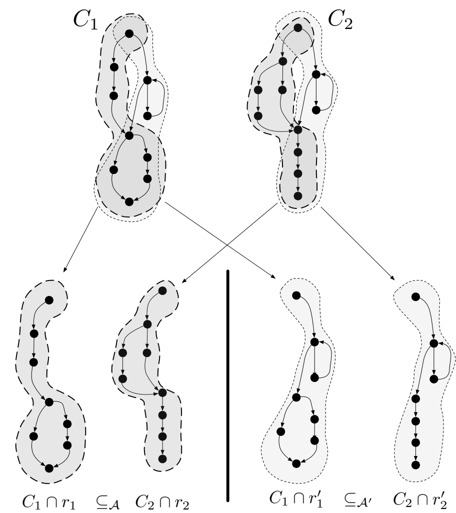


Fig. 1. Illustration of trace refinement relations.

The key idea is illustrated in Figure 1, where we are trying to show how a program C_1 relates to another program C_2, even though the programs may have different behaviors and even different atomic events. After abstracting the programs, we decompose C_1 into *trace classes*, each identified by a trace *restriction* r. In this example, there are two

trace classes within C_1, depicted with dashed-versus-dotted borders, and described using restrictions r_1 and r_1', respectively. These restrictions are applied with intersection: $C_1 \cap r_1$ and $C_1 \cap r_1'$. Next, for each such restriction on C_1 we identify a different restriction on C_2, in this case denoted r_2 and r_2', respectively. These restrictions help us focus on similarities, but there still may be differences between, for example $C_1 \cap r_1$ and $C_2 \cap r_2$: some atomic events may have different names or occur in one but not the other. To that end we identify a binary relation $\mathcal{A}$ relating traces of one program to traces of the other, in order to establish a relaxed inclusion $\subseteq_{\mathcal{A}}$ for the given trace classes, guided by that relation $\mathcal{A}$. We collect all of these $(r_1, r_2, \mathcal{A})$ triples to form what we call a *trace refinement relation* denoted $\mathbb{T}$. That is, we say that $\mathbb{T}$ is a refinement relation for C_1 and C_2, provided that

$$\forall (r_1, r_2, \mathcal{A}) \in \mathbb{T}.\ \ C_1 \cap r_1 \subseteq_{\mathcal{A}} C_2 \cap r_2 \quad \text{and} \quad C_1 \subseteq \bigcup_{(r_1, r_2, \mathcal{A}) \in \mathbb{T}} r_1.$$

The additional condition $C_1 \subseteq \bigcup_{(r_1, r_2, \mathcal{A}) \in \mathbb{T}} r_1$ requires that $\mathbb{T}$ identifies classes for all traces of C_1. Notice there is no particular requirement on the relationship between r_1 and r_2 within a tuple, affording flexibility in how to relate the corresponding classes. Completeness in this setting is essentially free: one can simply use binary relations $\mathcal{A}$ that ignore all the events in C_1. We therefore turn to more important questions such as what language can be used to specify trace classes, whether trace refinement relations can be found automatically, whether they capture our intuition about relationships between programs, and whether automation provides new insights about program relationships.

1.2 Challenges & Contributions

Specifying Trace Classes. To demonstrate trace refinement relations, we have instantiated the concept to a particular language for expressing traces: Kleene Algebra with Tests (KAT) [Kozen 1997]. We found KAT to be convenient because it is expressive, supports algebraic reasoning and has a built-in notion of composition. One could certainly also pursue temporal logics instead; we leave this to future work. KAT is an amalgamation of Kleene Algebra which (like regular expressions) has constructors for union +, concatenation ·, and star-iteration *, and Boolean Algebra which has boolean predicates and operations. (A background on KAT is given in Section 2.)

In Section 4 we instantiate trace refinement relations to KAT. We treat programs C_1 and C_2 in terms of their traces by abstracting them—via intermediate abstract programs—to KAT expressions, respectively denoted k_1 and k_2. In the setting of KAT, the $(r_1, r_2, \mathcal{A})$ tuples of $\mathbb{T}$ represent a KAT expression r_1 to restrict k_1 (via KAT intersection), a KAT expression r_2 to restrict k_2, and $\mathcal{A}$ is a set of KAT hypotheses, used to relate symbols/expressions in k_1 with symbols/expressions in k_2. Each tuple is a specific KAT inclusion and, as above, the overall trace refinement relation provides a weak, KAT-based notion of inclusion between the trace classes of C_1 and those of C_2.

In Section 4 we also present a formal development, defining trace refinement relations and putting them in the context of some other forms of refinement. We also show that, under this KAT instantiation, our trace-refinement relations are composable (Theorem 4.6). Composition permits an analysis of one pair of program fragments to be reused in many contexts and when the program is further changed [O'Hearn 2018], as discussed below.

Synthesizing Trace Refinement Relations. In Section 5 we describe methods for synthesizing a trace-refinement relation $\mathbb{T}$, given input programs C_1 and C_2. We first describe an overall algorithm that iteratively synthesizes $\mathbb{T}$ using a mixture of: (i) semantic abstraction α of a program C_1 (and C_2) to KAT expression k_1 (and k_2), (ii) pruning infeasible paths, (iii) symbolic KAT inclusion (via SymKAT [Pous 2015a]), (iv) a custom edit-distance algorithm on inclusion counterexamples to find relationships between cross-program trace classes, and (v) case-analysis on branching in C_1 and C_2

in circumstances where the branching in one program prevents immediate inclusion in the other. We have formalized our algorithm and proved that it is sound (Theorem 5.1).

The primary goal of our synthesized refinement relations is for them to be consumed by the algorithm/tool. To this end, we generate $\mathbb{T}$ to capture the multitude of conditions and ways in which one code fragment C_1 can relate to another fragment C_2, in order to produce results that are reusable. Once $\mathbb{T}$ is generated for one fragment pair, it can be reused in many contexts or even when the context code changes. This strategy is common in many program analysis tools (*e.g.* Facebook Infer). As a secondary goal, synthesized trace refinement relations can be used by experts. Like a syntactic diffing tool, the relations can guide them to understand how code has changed. Our tool, discussed next, is also geared toward generating output to meet this goal.

Implementation and Experimental Validation. In Section 7, we first discuss an implementation of our algorithm in a new tool called Knotical, that operates on an input pair of imperative `WHILE` programs and synthesizes trace refinement relations. Knotical is built from the ground up in OCaml using Interproc [Lalire et al. 2009] for abstract interpretation and Symkat to generate KAT counter examples [Pous 2015a, 2016]. We also describe a customized edit-distance implementation for scoring and finding alignments between programs.

We next describe an evaluation of our tool on a collection of 37 hand-crafted benchmark examples[1] that we have assembled. Almost all examples necessitate trace refinement relations that cannot be expressed using concrete refinement or other prior techniques. The examples range from those designed to exercise the various aspects of our approach (restriction, hypotheses, edit distance, etc.), to broader examples that model fragments of systems code behaviors, such as user I/O, array access patterns, and reactive web servers (*e.g.* `thttpd` [Poskanzer 2018] and Merecat [Nilsson 2019]). At the end of Section 7, we show that Knotical can synthesize trace refinement relations on these benchmarks; that these relations often capture our intuition about relationships between program pairs; and that these relations sometimes provide new insights as to unexpected relationships between program pairs. To enable human readable output, our tool ranks the solutions based on the number of hypotheses (fewer is better) and the replacement of same-type actions (e.g. a function call should not be replaced by an assignment). We found that this helped highlight which of the solutions indicate a closer correlation between the two programs.

1.3 Related Work and Limitations

Program equivalence and refinement are well-studied problems, and there are a wide range of formalisms and algorithmic techniques. Examples include concrete state refinement, state refinement relations, bisimulation and weak bisimulation. In Section 8, we set our work in the context of some of these prior works. We also discuss topics pertaining to reasoning about multiple traces within a single program such as hyper temporal logics [Clarkson et al. 2014] and self-composition [Barthe et al. 2004; Terauchi and Aiken 2005].

We developed a formal framework for trace refinement relations and, while KAT has worked well, it has also meant that we were restricted to terminating programs. We leave instantiation to other logics that support possibly non-terminating programs (*e.g.* temporal logic) to future work. Furthermore, although trace-refinement is particularly important in concurrency research, in this work we do not consider multiple threads. Instead, we use trace-refinement to more finely determine the similarities and differences between two programs. Our implementation was also limited in the number of symbols due to Symkat [Pous 2016]'s use of `char` to represent symbols. As noted in Section 5, soundness of our algorithm and tool depends on soundness of some of the subprocedures. Finally, the set of benchmarks we use were produced manually. We are not aware

[1] These examples can be found in the paper's artifact [Antonopoulos et al. 2019a,b].

Program C_1	**Program** C_2

<pre>1 while(x > 0) {
2   m = recv();
3   if (l) log(m);
4   if(m > 0) {
5   n = constructReply();
6   send(n);
7   if (l) log(n);
8   }
9   x--;
10 }</pre>
	<pre>1 while(x > 0) {
	2 m = recv();
	3 if (m > 0) {
	4 auth = check(m);
	5 if(auth > 0) {
	6 n = constructReply();
	7 send(n);
	8 }
	9 } else { log(m); }
	10 x--;
	11 }</pre>

Fig. 2. (Left) A simple reactive program C_1 that receives messages and sends replies. (Right) A modified version of C_2 with changes including the addition of authentication.

of a benchmark suite that fits our needs, and we had to manually validate our tool's solutions. We conclude in Section 9.1 by outlining future work toward adding support for heap-manipulating programs and, in Section 9.2, extensions for nested procedures and concurrency.

2 OVERVIEW

Consider the two programs in Figure 2, which we will use as a motivating example to highlight our approach. These two programs each model standard web-server behaviors, receiving requests and serving responses. The program C_2 is slightly different from C_1, possibly obtained as a modification to C_1, with added authentication and a new logging scheme.

We are interested in knowing how the new program C_2 compares to the previous program C_1. (We take a reactive view of programs and, for simplicity, just work with stack variables and events—denoted **recv**, **log**, etc.—corresponding to I/O side effects, function calls, etc.) Both of these programs involve typical web server behavior: alternately receive a request and send a response. The programs involve some differences, arising from changes/edits that were made to C_1. Yet there are still similarities: both programs involve a loop that iterates over x, **recv**ing messages and possibly **send**ing responses. On the other hand, C_2 only performs a **log** when it **recv**s an m such that m $\leq$ 0, and it additionally performs an authorization **check** on m. In addition, C_1 only performs **log**s when the flag l is enabled.

We would like to express *similarities* in how the programs behave over time, such as alternation between **send** and **recv**. We would like a formal framework to also tolerate the *differences* between how the programs behave over time, such as the **recv**/**send** behavior in C_1 versus the **recv**/**check**/**send** behavior in C_2. Intuitively, the formal framework we develop will need some way of expressing different behaviors of one program (*e.g.* auth is always greater than 0 in C_2) that correlate to behaviors in the other (*e.g.* **log**-free traces of C_1). Moreover, we need an abstract way to relate an event in one program (*e.g.* the **send** event in C_1) to an analogous event in the other program (*e.g.* **send** in C_2).

In this paper we describe a way to specify these kinds of "weak" trace relationships. Beyond this example, there are many intuitive program pair properties along these lines including (i) *the new program alternately receives and sends messages like the original*, (ii) *the new program additionally performs an authorization check after each receive*, (iii) *the new program's memory pool usage pattern matches the old program's memory allocation pattern*, and so on. These properties are trace based (as opposed to pre/post) and also relational, yet weaker than trace equivalence. We aren't aware of existing works aimed at these properties, for example by using ghost variables to encode traces into the relations of Benton [2004] / Yang [2007] or by weakening bisimulation in this direction.

In the setting of understanding these kinds of similarities/differences between programs, it is difficult to define a particular "property" of interest. Instead, we are after *relationships* between (the traces of) a pair of programs. In this paper we will express these relationships formally in an appropriate language. We will now discuss our choice of language for describing these relationships.

2.1 Background: From Programs to KAT Expressions

Expressing properties of the way a program behaves over time motivates the need for a suitable trace specification language. For the purposes of this paper, we focus on traces described using Kleene Algebra with Tests (KAT) [Kozen 1997]. KAT is a convenient choice because it is expressive, supports algebraic reasoning, and has a built-in composition operator. We leave exploration of trace refinement relations in other languages (*e.g.* temporal logic) to future work.

KAT Reminder. We briefly recall KAT here and provide a formal definition in Section 3. KAT is an amalgamation of Kleene Algebra which (like regular expressions) has constructors for union $+$, concatenation $\cdot$, and star-iteration $*$, and Boolean Algebra which has boolean predicates and operations. KAT expressions consist of a combination of event symbols (herein denoted in uppercase: A, B, C) and boolean "test" symbols (herein denoted in lowercase: a, b, c). One can write KAT expressions that mix event symbols with boolean test symbols, *e.g.* $(\mathsf{a}\cdot\mathsf{A})^*\cdot\mathsf{C}\cdot\mathsf{b}$. KAT expressions allow for an easy-to-use representation of `while`-style imperative programs [Kozen 1997, 2006] (see also Section 4.1). The *union* and *Kleene star* operations of Kleene Algebra correspond to conditionals and while loops in the program, and the underlying Boolean Algebra helps with encoding under which boolean conditions a path can materialize. For example, the KAT expression $(\mathsf{b}\cdot\mathsf{C} + \overline{\mathsf{b}}\cdot\mathsf{D})^*$ models a program that is a multi-path loop, branching on b. Symbol b is an element of the Boolean subalgebra, while symbols C and D correspond to the respective actions in the program.

Returning to the example in Figure 2, we introduce a KAT geared specifically for this program pair. For notational convenience, we write boolean test symbols such as "$\mathsf{a}_{\mathtt{x>0}}$" for integer expressions such as `x>0`, using subscripts to indicate which program expressions correspond to the symbols. Similarly, we write event symbols for program statements such as "$\mathsf{E}_{\mathbf{recv}}$" for action **recv**. The behaviors of programs C_1 and C_2 can be represented, respectively, as:

$$
\begin{aligned}
k_1 \triangleq\ & (\mathsf{a}_{\mathtt{x>0}}\cdot(\mathsf{E}_{\mathbf{recv}}\cdot(\mathsf{b}_{\mathtt{l=true}}\cdot\mathsf{O}_{\mathbf{log}} + \overline{\mathsf{b}_{\mathtt{l=true}}}\cdot 1)\cdot(\mathsf{c}_{\mathtt{m>0}}\cdot\mathsf{C}_{\mathbf{Rep}}\cdot\mathsf{S}_{\mathbf{send}} \\
& \quad (\mathsf{b}_{\mathtt{l=true}}\cdot\mathsf{L}_{\mathbf{log}} + \overline{\mathsf{b}_{\mathtt{l=true}}}\cdot 1) + \overline{\mathsf{c}_{\mathtt{m>0}}}\cdot 1)\cdot\mathsf{X}_{\mathbf{x--}}))^*\cdot\overline{\mathsf{a}_{\mathtt{x>0}}} \\
k_2 \triangleq\ & (\mathsf{a}_{\mathtt{x>0}}\cdot(\mathsf{E}_{\mathbf{recv}}\cdot(\mathsf{c}_{\mathtt{m>0}}\cdot\mathsf{K}_{\mathbf{check}}\cdot(\mathsf{d}_{\mathtt{auth>0}}\cdot\mathsf{C}_{\mathbf{Rep}}\cdot\mathsf{S}_{\mathbf{send}} \\
& \quad + \overline{\mathsf{d}_{\mathtt{auth>0}}}\cdot 1) + \overline{\mathsf{c}_{\mathtt{m>0}}}\cdot\mathsf{O}_{\mathbf{log}})\cdot\mathsf{X}_{\mathbf{x--}})^*\cdot\overline{\mathsf{a}_{\mathtt{x>0}}}
\end{aligned}
$$

where "1" is the identity symbol in KAT, akin to `skip` in programs. ("0" in KAT is akin to `assume(false)`.) Note that composition $\cdot$ binds tighter than union $+$, and we use overline (*e.g.* $\overline{\mathsf{c}_{\mathtt{m>0}}}$) to indicate negation. The above KAT symbols represent program statements, but we use KAT symbols more generally as semantic entities. KAT also can express many well-known tricks for increasing path sensitivity such as loop unrolling, trace partitioning [Mauborgne and Rival 2005], control-flow refinement [Gulwani et al. 2009]. For simplicity, we omit such examples.

2.2 Trace Refinement Relations

Not all behaviors of C_1 relate to behaviors of C_2 (such as some logging in C_1) and vice-versa (authorization failures in C_2), so exact trace equivalence does not hold for the two programs in Figure 2. Nevertheless, we are still interested in *which* trace classes of C_1 are also related to trace classes of C_2 and how one might correlate events in C_1 with those in C_2. We wish to describe, for example, how both programs have a substantially similar **recv**/**send** relationship. Imagine that we could somehow focus on the behaviors of C_2 in which `auth` was always greater than 0, somehow

focus on the behaviors of C_1 that had no **log** events (when l was always false), and *also* on the executions of both programs where they **recv** valid messages and thus m > 0. In that case, the programs would have *restricted* behaviors, represented as the following restricted KAT expressions:

$$(\mathsf{a}_{\mathtt{x>0}} \cdot (\mathsf{E}_{\mathbf{recv}} \cdot (\mathsf{c}_{\mathtt{m>0}} \cdot \mathsf{C}_{\mathbf{Rep}} \cdot \mathsf{S}_{\mathbf{send}}) \cdot \mathsf{X}_{\mathbf{x--}}))^* \cdot \overline{\mathsf{a}_{\mathtt{x>0}}} \leq k_1 \tag{1}$$

$$(\mathsf{a}_{\mathtt{x>0}} \cdot (\mathsf{E}_{\mathbf{recv}} \cdot (\mathsf{c}_{\mathtt{m>0}} \cdot \mathsf{K}_{\mathbf{check}} \cdot \mathsf{C}_{\mathbf{Rep}} \cdot \mathsf{S}_{\mathbf{send}}) \cdot \mathsf{X}_{\mathbf{x--}}))^* \cdot \overline{\mathsf{a}_{\mathtt{x>0}}} \leq k_2 \tag{2}$$

The above equations are just *classes* of trace behaviors of C_1 and C_2, respectively, with ≤ denoting KAT inclusion. If we could now further somehow *ignore* the $\mathsf{K}_{\mathbf{check}}$ event in C_2, the above KAT expressions would be equivalent. (In this case they are syntactically equivalent, but they could also be semantically equivalent.) Finding this correlation takes care of some behaviors of C_1, but we aim to exhaustively consider all of its behaviors.

The idea of relating specific behaviors between programs is the first step toward our notion of trace refinement relations, formalized in Section 4. We consider one class of traces of C_1 at a time like we did above in Equations 1 and 2. More precisely, we use a KAT expression r_1 to represent a trace *restriction* and KAT intersection $k_1 \cap r_1$ to represent a (restricted) trace class. For this restricted behavior of k_1, it is typically helpful to restrict k_2 (which corresponds to C_2) with a perhaps rather unrelated r_2. Then we can ask whether equivalence holds between $k_1 \cap r_1$ and $k_2 \cap r_2$. Returning to the running example, we can consider the class of traces of C_1 that do not involve logging by letting

$$r_1 = (\mathsf{a}_{\mathtt{x>0}} \cdot \boxed{\overline{\mathsf{b}_{\mathtt{l=true}}}} \cdot \mathsf{Any})^* \cdot \overline{\mathsf{a}_{\mathtt{x>0}}}, \tag{3}$$

requiring that $\overline{\mathsf{b}_{\mathtt{l=true}}}$ holds between any actions while $\mathsf{a}_{\mathtt{x>0}}$ also holds. Here Any is shorthand for the disjunction of all event symbols in the KAT at hand. We can use this restriction to focus on $k_1 \cap r_1$. Similarly we can restrict C_2 to the class of traces in which auth is always above 0 by letting

$$r_2 = (\mathsf{a}_{\mathtt{x>0}} \cdot \boxed{\mathsf{c}_{\mathtt{m>0}}} \cdot \boxed{\mathsf{d}_{\mathtt{auth>0}}} \cdot \mathsf{Any})^* \cdot \overline{\mathsf{a}_{\mathtt{x>0}}} \tag{4}$$

This restriction allows only behaviors of the program where both m > 0 and auth > 0 hold between any actions while $\mathsf{a}_{\mathtt{x>0}}$ also holds. With these restrictions in place, we get Equations 1 and 2 above.

In some cases, we can witness classes of traces in C_1 that are a part of trace classes in C_2 simply using such a pair of restrictions. However, restrictions are not the only way that we relate k_1 to k_2. Looking at Equations 1 and 2, there is still the discrepancy that the $\mathsf{K}_{\mathbf{check}}$ event occurs in C_2 but not C_1. Since we are already focused on a case where auth is always greater than 0, the $\mathsf{K}_{\mathbf{check}}$ event is not so important. We can ignore such unimportant events by introducing additional *hypotheses* $\mathcal{A}$ into the KAT. In this case, we can introduce the hypothesis $\mathcal{A} = \{\mathsf{K}_{\mathbf{check}} = 1\}$, and we finally have the KAT relationship $(k_1 \cap r_1) =_{\{\mathsf{K}_{\mathbf{check}}=1\}} (k_2 \cap r_2)$. KAT enables us to exploit algebraic reasoning and so we can introduce hypotheses for other purposes too. It is often convenient to let syntactically identical statements between C_1 and C_2 use the same KAT symbol. In other cases, we may prefer not to, but we can introduce KAT hypotheses to instead selectively relate statements.

Putting It All Together. Ultimately, we will collect a set $\mathbb{T} = \{(r_1, r_2, \mathcal{A}), (r_1', r_2', \mathcal{A}'), \ldots\}$, each tuple considering a different way in which a trace class of C_1 relates to a trace class of C_2. We finally require the union over the first projection of $\mathbb{T}$ to ensure that all of C_1 is represented via trace classes. In Section 4 we formalize this as a *trace refinement relation.* Notice that "weak" completeness is straight-forward: we could always add a triple $(1, 1, \mathcal{A}^\top)$ where $\mathcal{A}^\top$ maps every single symbol to 1 (skip). The challenges instead include whether trace-refinement relations can be generated automatically, whether they confirm our intuition about the relationship between program pairs, and whether they can provide new insights about program pair relationships.

2.3 Composition, Contexts, Spanning Versions

So far we have discussed reasoning about a change from C_1 to C_2, while the context remains fixed. But what about the context? A single fragment C_1 can be used in many different contexts within a large program. Thus, when C_1 is changed to C_2, there is benefit to performing a single analysis that considers all possible contexts, rather than considering how C_1 and C_2 relate in each context [O'Hearn 2018]. This approach also allows us to cope with the fact that the context *itself* may change.

Returning to Figure 2, fragments C_1 and C_2 may be used in different contexts. Perhaps in one context it is important that *all failed connections are logged* and we want to ensure a change from C_1 to C_2 preserves this property. In that case we need a refinement relation that does not ignore **log** events, and assume that the context of C_1 ensures $\mathsf{l} = \mathsf{true}$. Formally, we would have the tuple

$$((\mathsf{b}_{\mathsf{l}=\mathsf{true}} \cdot \overline{\mathsf{c}_{\mathsf{m}\rhd 0}} \cdot \mathsf{Any})^*, (\overline{\mathsf{c}_{\mathsf{m}\rhd 0}} \cdot \mathsf{Any})^*, \mathcal{A}_{log}) \in \mathbb{T}$$

This restricts to trace classes of C_1 where logging is enabled and all connections fail and restricts traces of C_2 to those where all connections fail. Moreover, we require a set of hypotheses $\mathcal{A}_{log}$ which does not imply that $\mathsf{O}_{\mathbf{log}} = 1$. In a different context, other relations would be relevant. As noted earlier, this example comes from a change that added SSL support to `thttpd` [Poskanzer 2018]. Therefore, we may wish to have a refinement relation, specifying that *as long as all messages are authenticated in* C_2*, then it behaves the same as* C_1.

$$(\mathsf{Any}^*, (\mathsf{d}_{\mathsf{auth}\rhd 0} \cdot \mathsf{Any})^*, \{\mathsf{O}_{\mathbf{log}} = 1, \mathsf{L}_{\mathbf{log}} = 1, \mathsf{K}_{\mathbf{check}} = 1\}) \in \mathbb{T}$$

Here, C_1 is unrestricted, C_2 is focused only on executions that are authenticated, and we use a set of hypotheses that ignores all **log** events and ignores the **check** event in k_2.

Our formalism can capture other more complicated contexts, such as an outer loop. We have proved that our trace-refinement relations are *compositional* (Theorem 4.6), across all of KAT, allowing us to reason about the overall trace-refinement, by considering pairs of program segments at a time.

2.4 Automation

In Section 5 we describe an algorithm for synthesizing trace refinement relations. At the high level, each iteration of the algorithm is a recursive call (Synth), where we are exploring a region of the solution space where C_1 has possibly been r_1-restricted, C_2 has possibly been r_2-restricted, and a collection of KAT hypotheses $\mathcal{A}$ are in use. Each iteration of our algorithm calculates KAT abstractions k_1 and k_2 from programs $C_1 \cap r_1$ and $C_2 \cap r_2$, resp., and then considers whether k_1 is included in or is equivalent to k_2, under the current set $\mathcal{A}$ of hypotheses. To check this refinement, we use Symkat [Pous 2016]. If this refinement holds, then the algorithm returns this triple $(k_1, k_2, \mathcal{A})$ as a solution that may be assembled with others into a complete solution by previous calls to Synth. Alternatively, if the inclusion/equivalence does not hold, we decide, based partly on the counterexamples, whether to (i) introduce restrictions $(r_{1,i}, r_{2,i})$ and/or (ii) introduce hypotheses $\mathcal{A}_i$. In Sections 5 and 6 we discuss how the sub-procedure employs a custom edit distance algorithm for this purpose. Finally, the restrictions are instrumented back into the programs, to produce new programs that are considered recursively.

Naturally, our algorithm needs to build on a method for translating back and forth between a program C and its corresponding KAT expression k. The former lets us learn fine-grained details about the behavior of the program, while the latter lets us perform coarse-grained cross-program comparisons. To get k from C, we exploit program semantics to obtain precise KAT expressions, for example by excluding infeasible paths. Our abstraction is not a one-way process: our algorithmic

search involves discovering restrictions r and then instrumenting them back into the source program, using a form of product program.

We have proved that our algorithm is sound (Theorem 5.1) meaning that a trace refinement relation $\mathbb{T}$ returned by our algorithm is indeed one that satisfies $k_1 \preceq^{\mathbb{T}} k_2$, for k_1 and k_2 being the KAT expressions corresponding respectively to the input programs C_1 and C_2. Weak completeness is less interesting; we are instead interested in *automatically* generating trace-refinement relations and whether these relations capture our intuition and/or provide new insights into the relationships between program pairs.

2.5 The Knotical Tool

We have developed a prototype tool Knotical that implements our algorithm. During the algorithm, when considering whether the current k_1 refines k_2, Symkat may find that it does not and return a counterexample of a string w_1 that is in k_1 but not k_2 (and w_2, vice-versa). Our algorithm departs from a traditional counterexample-guided approach and instead is able to consider not only the entirety of counterexample strings w_1 and w_2, but also the KAT expressions k_1 and k_2, in order to find a better correlation between the two. It is easy for a human reader to see that the relationship between k_1 and k_2 fits better, when the * expression in k_1 is correlated with the * expression in k_2. To this end, we developed a custom edit-distance algorithm [Bille 2005] (see Section 6).

We are not aware of any existing benchmarks that would be appropriate for evaluating Knotical since prior works are focused mainly on state-based relationships. We created a series of 37 small benchmarks (Section 7), including examples that model user I/O, array access patterns, and fragments of reactive web servers (*e.g.* thttpd [Poskanzer 2018] and Merecat [Nilsson 2019]). On most benchmarks, our tool was able to generate a trace-refinement relation in seconds or fractions of a second. These relations often capture human intuition about relationships between programs and sometimes provide new insights as to how the programs relate.

3 PRELIMINARIES

Strings, Sets, Composition, Programs. A string s over an alphabet Σ is a sequence $s_1 \cdot s_2 \cdots s_n$ of symbols $s_i \in \Sigma$, for $i \in [1, n]$. Given sets $S_1, \ldots, S_n$, a set $S \subseteq S_1 \times S_2 \times \ldots S_n$, and an element $s = (s_1, \ldots, s_n) \in S$ we denote with $\mathsf{proj}_i(s)$ the projection of s to its i-th element s_i in S_i. We abuse notation, denoting as $\mathsf{proj}_i(S)$ the set $\{s_i \in S_i \mid s_i \in \mathsf{proj}_i(s), s \in S\}$.

We assume a set Prog of (essentially imperative) *programs* operating on a set $\mathcal{S}$ of *states*. We assume a distinguished "error state" fault $\in \mathcal{S}$. A *configuration* is a pair $\langle C, \sigma \rangle$, where C is a program and σ a state; we write Config for the set of all configurations. We assume a binary relation $\rightsquigarrow \subseteq \text{Config} \times \mathcal{S}$ capturing the "big step", nondeterministic operational semantics of our programs; $\langle C, \sigma \rangle \rightsquigarrow \rho$ means that executing program C in initial state σ can result in the final state ρ.

Kleene Algebra with Tests. We use KAT [Kozen 1997] to represent classes of traces within a program. A *Kleene Algebra with Tests* $\mathcal{K}$ is a two-sorted structure $(\Sigma, \mathcal{B}, +, \cdot, ^*, ^-, 0, 1)$, where $(\Sigma, +, \cdot, ^*, 0, 1)$ is a Kleene algebra, $(\mathcal{B}, +, \cdot, ^-, 0, 1)$ is a Boolean algebra, and $(\mathcal{B}, +, \cdot, 0, 1)$ is a subalgebra of $(\Sigma, +, \cdot, 0, 1)$. We distinguish between two sets of symbols: set P for primitive actions, and set B for primitive tests. The grammar of boolean test expressions is: $\mathsf{BExp} ::= b \in \mathsf{B} \mid b_1 \cdot b_2 \mid b_1 + b_2 \mid \bar{b} \mid 0 \mid 1$ and we define the grammar KExp of KAT expressions as:

$$\mathsf{KExp} ::= p \in \mathsf{P} \mid b \in \mathsf{BExp} \mid k_1 \cdot k_2 \mid k_1 + k_2 \mid k^* \mid 0 \mid 1$$

The free Kleene algebra with tests over $\mathsf{P} \cup \mathsf{B}$, is obtained by quotienting BExp with the axioms of Boolean algebras, and KExp with the axioms of Kleene Algebra. For $e, f \in \mathcal{K}$, we write $e \leq f$ if $e + f = f$, and all Kleene Algebras with Tests $\mathcal{K}$ we consider here are $*$-continuous, where any

elements a, b, c in $\mathcal{K}$, satisfy the axiom $a \cdot b^* \cdot c = \sum_{n \in \mathbb{N}} a \cdot b^n \cdot c$ [Kozen 1990]. If the relationship $e \leq f$ holds under a set of hypotheses $\mathcal{A}$, we write $e \leq_{\mathcal{A}} f$. By convention we use lower case letters for test symbols and upper case letters for actions. We may also abuse notation, writing program conditions and statements rather than boolean symbols and action symbols (in which case we implicitly create symbols for each). For Figure 2 booleans include $\mathsf{B} = \{\mathsf{a}_{x>0}, \mathsf{b}_{1=\mathsf{true}}\}$, actions include $\mathsf{P} = \{\mathsf{O}_{\mathbf{log}}, \mathsf{E}_{\mathbf{recv}}\}$, and $k = \ldots(\mathsf{b}_{1=\mathsf{true}} \cdot \mathsf{O}_{\mathbf{log}} + \overline{\mathsf{b}_{1=\mathsf{true}}} \cdot 1)\ldots \in \mathcal{K}$.

Definition 3.1 (Intersection). Given a KAT $\mathcal{K}$ and two of its elements k_1 and k_2 we define $k_1 \cap k_2$ to be equal to $l_1 + \ldots + l_n + \ldots$, where $\{l_i\}_{i \in \mathbb{N}}$ is the set of all elements l_i in $\mathcal{K}$ such that $l_i \leq k_1$ and $l_i \leq k_2$.[2]

For KAT expressions k_1, k_2 and l, and a set of hypotheses $\mathcal{A}$, we write $l \in k_1 \setminus_{\mathcal{A}} k_2$ if $l \leq_{\mathcal{A}} k_1$ and $l \not\leq_{\mathcal{A}} k_2$. Similarly, we write $l \in k_1 \Delta_{\mathcal{A}} k_2$ if $l \in k_1 \setminus_{\mathcal{A}} k_2$ or $l \in k_2 \setminus_{\mathcal{A}} k_1$ (akin to symmetric difference on sets). Finally, for two KATs $\mathcal{K}_1$ and $\mathcal{K}_2$, we denote with $\mathcal{K}_1 \cup \mathcal{K}_2$ the smallest KAT that contains both $\mathcal{K}_1$ and $\mathcal{K}_2$. Finally, when we refer to strings we mean KAT strings, which are KAT expressions where only the concatenation operation is used.

Program Refinement. Program refinement is a classical concept and can be formulated in different ways, depending on the context. Often, the usual notion of refinement is too concrete because it does not consider the context in which C_1 and C_2 are used. Benton [2004] and Yang [2007] introduced a weaker notion of refinement, parameterized by an input relation between the states of the two programs as well as an output relation. We call this an *interface*, which is an equivalence relation on the set of states $\mathcal{S}$ and defined as follows:

Definition 3.2. For interfaces I, O and programs C, C', we say C' *refines* C *w.r.t.* (I, O), written $C' \leq^I_O C$, if the following two conditions are met, for all states σ, σ' such that $I(\sigma, \sigma')$:

(1) if $\langle C', \sigma' \rangle \rightsquigarrow$ fault, then $\langle C, \sigma \rangle \rightsquigarrow$ fault;
(2) if $\langle C', \sigma' \rangle \rightsquigarrow \rho'$, then either there exists ρ such that $\langle C, \sigma \rangle \rightsquigarrow \rho$ and $O(\rho, \rho')$, or else $\langle C, \sigma \rangle \rightsquigarrow$ fault.

We say that C' *(concretely) refines* C, written $C' \leq C$, when $C' \leq^{\mathsf{id}}_{\mathsf{id}} C$ where id is the *identity relation.*[3]

4 KAT REPRESENTATIONS AND REFINEMENTS

In this section we discuss a two-step semantic abstraction (Section 4.1), trace refinement and trace-refinement relations (Section 4.2), and composition results (Section 4.3).

4.1 Abstracting Programs into KAT Expressions

We describe how to abstract a while-style program C to a KAT expression k over a KAT $\mathcal{K}$. We parameterize such a translation by an abstraction α used for both abstracting concrete states of the program to abstract states, as well as the latter to elements of the boolean subalgebra of $\mathcal{K}$. More concretely, given a program C over a set of states $\mathcal{S}$, we define α to be a tuple $(\mathcal{K}, A_S, \alpha_S, \alpha_B)$, where $\mathcal{K}$ is a KAT, A_S is a set of abstract states, α_S is a mapping from $\mathcal{S}$ to A_S corresponding to the program abstraction given by the abstract interpretation, and α_B is a mapping from A_S to $\mathcal{B}$,

[2] For the interested reader, this definition agrees with other definitions regarding intersection, namely ones defined on the set of guarded strings associated with a KAT expression, *i.e.* $[\![k_1 \cap k_2]\!] = [\![k_1]\!] \cap [\![k_2]\!]$. See Theorem A.1 in the Appendix. Notice that for any two KAT expressions $k_1 + \ldots + k_n$ and $l_1 + \ldots + l_m$ over a KAT $\mathcal{K}$, for $n, m \in \mathbb{N}$, there is a finite number of elements $h_1 + \ldots + h_r$, for $r \in \mathbb{N}$ such that $(k_1 + \ldots + k_n) \cap (l_1 + \ldots + l_m) = h_1 + \ldots + h_r$. Since we never start with a KAT expression as an infinite disjunction in what follows, any time we talk about the intersection of two KAT expressions as a disjunction of KAT elements, we will refer to such a finite disjunction.

[3] Benton used the notation $\vdash C \sim C' : I \Rightarrow O$ whereas we use notation by James Brotherston (personal communication).

the boolean subalgebra of $\mathcal{K}$. Additionally, we require that for any $b \in \mathcal{B}$, there is a set of states $\{a_1, \ldots, a_n\} \in A_S$ such that $b = \alpha_B(a_1) + \ldots + \alpha_B(a_n)$. When $\mathcal{K}$ and A_S are clear from the context, we write $\alpha = \alpha_B \circ \alpha_S$.

With such an abstraction $\alpha = (\mathcal{K}, A_S, \alpha_S, \alpha_B)$ as a parameter, we say a translation from C to a KAT expression $k \in \mathcal{K}$ is *valid* (resp. *strongly valid*), if for any states $\sigma, \rho \in \mathcal{S}$, $\langle C, \sigma\rangle \rightsquigarrow \rho$ only if (resp. if and only if) $\alpha_B(\alpha_S(\sigma)) \cdot k \cdot \alpha_B(\alpha_S(\rho)) \neq 0$. We assume a procedure $\textsc{Translate}(C, \alpha)$ that returns $k \in \mathcal{K}$ and the translation from C to k is valid (see Section 5 for an implementation). Finally, we will iteratively construct abstractions and thus need the following notion of refinement over abstractions:

Definition 4.1 (Refining abstractions). For two given abstractions $\alpha = (\mathcal{K}, A_S, \alpha_S, \alpha_B)$ and $\alpha' = (\mathcal{K}', A'_S, \alpha'_S, \alpha'_B)$ over the same set of concrete states $\mathcal{S}$, we say that α' *refines* α, and write it as $\alpha' \sqsubseteq \alpha$, if $\mathcal{K}$ is a subalgebra of $\mathcal{K}'$ and for any state $\sigma \in \mathcal{S}$, $\alpha'_B(\alpha'_S(\sigma)) \leq \alpha_B(\alpha_S(\sigma))$.

Let $\alpha_1 = (\mathcal{K}_1, A^1_S, \alpha^1_S, \alpha^1_B)$ and $\alpha_2 = (\mathcal{K}_2, A^2_S, \alpha^2_S, \alpha^2_B)$ be two abstractions, both refining an abstraction α with Boolean algebra $\mathcal{B}$. By $\alpha^1_S \times \alpha^2_S$ we denote the function from $\mathcal{S}$ to $A^1_S \times A^2_S$, that maps a state $\sigma \in \mathcal{S}$ to $(\alpha^1_S(\sigma), \alpha^2_S(\sigma))$. Further, we define $\alpha^1_B \cdot \alpha^2_B$ to be the function from $A^1_S \times A^2_S$ to $\mathcal{B}$ that maps a tuple $(a_1, a_2) \in A^1_S \times A^2_S$ to $\alpha^1_B(a_1) \cdot \alpha^2_B(a_2)$ in $\mathcal{B}$. The *combined abstraction* of α_1 and α_2, written $\alpha_1 \sqcup \alpha_2$, is defined to be the abstraction $(\mathcal{K}_1 \cup \mathcal{K}_2, A^1_S \times A^2_S, \alpha^1_S \times \alpha^2_S, \alpha^1_B \cdot \alpha^2_B)$.

4.2 KAT Refinements

With abstractions from programs to KAT expressions in hand, we now first define *concrete* KAT refinement, and then our notion of trace-refinement *relations* (Definition 4.4).

Definition 4.2 (Concrete KAT refinement). Let k_1 and k_2 be two KAT expressions over $\mathcal{K}$. We say that k_1 *concretely refines* k_2, and denote it by $k_1 \preceq k_2$, if for any $b, d \in \mathcal{B}$:

(1) $b \cdot k_1 = 0$ implies $b \cdot k_2 = 0$,
(2) $b \cdot k_1 \cdot d \neq 0$ implies $b \cdot k_2 \cdot d \neq 0$, or $b \cdot k_2 = 0$.

The following relates concrete trace refinement, via abstraction, back to concrete program refinement (See Appendix A).

Theorem 4.3. *Let C_1 and C_2 be two programs, and let k_1 and k_2 be the two KAT expressions obtained from a strongly valid translation of the two programs respectively, under some abstraction α. Then it holds that $C_1 \preceq C_2$ if and only if $k_1 \preceq k_2$.*

Notice that for the inclusion implication to work in both directions we need the translations to be strongly valid. We currently do not enforce this in our implementation, both because the underlying tools we employ (Interproc [Lalire et al. 2009]) would not be able to satisfy this in general, but also because it is a theoretically hard requirement, unless restricted to limited classes of programs. We simply require a valid translation in our implementation, where we reason about the program pairs using the KAT domain (*i.e.* a relation over abstractions).

We now weaken concrete KAT refinement, presenting trace-refinement relations. Intuitively, the idea is to reason piece-wise, considering classes of traces within k_1 and, for each, correlating them with a corresponding trace class in k_2, with the help of KAT hypotheses. Note that, for some element k of a KAT $\mathcal{K}$, we say a set $S = \{s_1, \ldots, s_n\}$ of $\mathcal{K}$ elements *partitions* k, if $k = s_1 + \ldots + s_n$.

Definition 4.4 (Trace Refinement Relations). Let $\mathcal{K}$ be a KAT, let $\mathfrak{A}$ be a class of hypotheses over $\mathcal{K}$, and let $\mathbb{T}$ be a relation over $\mathcal{K} \times \mathcal{K} \times \mathcal{P}(\mathfrak{A})$. Given two KAT elements k_1 and k_2 of $\mathcal{K}$, we say that k_1 *refines* k_2, with respect to $\mathbb{T}$, denoted by $k_1 \preceq^{\mathbb{T}} k_2$, if $\text{proj}_1(\mathbb{T})$ partitions k_1 and,

$$\text{for any } (l_1, l_2, \mathcal{A}) \in \mathbb{T}, \qquad l_1 \cap k_1 \preceq_{\mathcal{A}} l_2 \cap k_2.$$

We also consider *trace equivalence relations*, slightly adapting Definition 4.4 to use equivalence (=), rather than inclusion ($\leq$), as well as requiring that both $\text{proj}_1(\mathbb{T})$ partitions k_1 and $\text{proj}_2(\mathbb{T})$ partitions k_2.

As discussed in Section 2, intuitively each $(l_1, l_2, \mathcal{A})$ triple in a trace-refinement relation $\mathbb{T}$ identifies restrictions on k_1 and k_2, as well as KAT hypotheses $\mathcal{A}$ that allow us to align the $k_1 \cap l_1$ trace classes with ones in $k_2 \cap l_2$. In the example from Section 2, we gave examples of an l_1 that excluded logging by forcing $\overline{\mathsf{b}_{1=\mathsf{true}}}$ to hold at each iteration of the loop.

Remark 4.5. *As trace-refinement is a weakening of concrete refinement, it is natural that two KAT expressions k_1 and k_2 may be such that k_1 refines k_2, but does not concretely refine it. For most expressions k_1 and k_2, we can find a set of hypotheses $\mathcal{A}$ (e.g. that equates all actions and specific boolean variables to* 1*), such that the singleton set containing only the tuple* $(1, 1, \mathcal{A})$ *is a trivial solution to trace-refinement between k_1 and k_2.*

Finally, we overload the KAT refinement definition to be used on programs themselves, when the abstraction α is clear from the context. Thus, for two programs C_1 and C_2, and a trace-refinement relation $\mathbb{T}$, we may write $C_1 \leq^{\mathbb{T}} C_2$ to mean that $\text{Translate}(C_1, \alpha) \leq^{\mathbb{T}} \text{Translate}(C_2, \alpha)$.

Classes of Hypotheses. For this work, we will explore the effect of just a few types of classes of hypotheses. In general, checking equality of KAT expressions under arbitrary additional hypotheses, can be undecidable ([Kozen 1996]). Because of that, and guided by the limitations imposed by certain libraries we use in our implementation (Symkat), we focus on the following types of hypotheses when $\mathsf{A}, \mathsf{B} \in \mathsf{P}$ and $\mathsf{a}, \mathsf{b} \in \mathsf{B}$: (i) to ignore certain actions: $\mathsf{A} = 1$, (ii) to fix the valuation of certain booleans: $\mathsf{b} = 1$ or $\mathsf{b} = 0$, (iii) to express commutativity of actions against tests: $\mathsf{A} \cdot \mathsf{b} = \mathsf{b} \cdot \mathsf{A}$ (currently not used in our implementation) and (iv) to relate single elements: $\mathsf{A} = \mathsf{B}$ or $\mathsf{a} = \mathsf{b}$.

4.3 Composition

Given trace-refinement relations $\mathbb{T}_1$ and $\mathbb{T}_2$, we define their *composition* $\mathbb{T}_1 \odot \mathbb{T}_2$ to be the trace-refinement relation $\mathbb{T} = \{(l_1 \cdot m_1, l_2 \cdot m_2, \mathcal{A}_1 \cup \mathcal{A}_2) \mid (l_1, l_2, \mathcal{A}_1) \in \mathbb{T}_1, (m_1, m_2, \mathcal{A}_2) \in \mathbb{T}_2\}$. Similarly, we define *disjunction* $\mathbb{T}_1 \oplus \mathbb{T}_2$ to be the trace-refinement relation $\mathbb{T} = \{(l_1 + m_1, l_2 + m_2, \mathcal{A}_1 \cup \mathcal{A}_2) \mid (l_1, l_2, \mathcal{A}_1) \in \mathbb{T}_1, (m_1, m_2, \mathcal{A}_2) \in \mathbb{T}_2\}$. Finally, for $\mathbb{T}$, we define $\mathbb{T}^\star$ to be $\{(o^*, p^*, \mathcal{A}) \mid (o, p, \mathcal{A}) \in \mathbb{T}\}$.

Theorem 4.6 below allows us to reason about individual fragments of KAT expressions, and combine the analyses into a result that holds overall. We can do so by building trace-refinement relations in a bottom-up fashion, capturing larger and larger fragments of those KAT expressions, guided by their structure. (See Appendix A)

Theorem 4.6. *Suppose k_1, k_2, l_1 and l_2 are KAT expressions. Let $\mathbb{T}_k$ and $\mathbb{T}_l$ be trace-refinement relations, such that $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$. Then $k_1 \cdot l_1 \leq^{\mathbb{T}_k \odot \mathbb{T}_l} k_2 \cdot l_2$, $k_1 + l_1 \leq^{\mathbb{T}_k \oplus \mathbb{T}_l} k_2 + l_2$, $k_1 + l_1 \leq^{\mathbb{T}_k \cup \mathbb{T}_l} k_2 + l_2$, and $k_1^* \leq^{\mathbb{T}_k^\star} k_2^*$.*

As a simple corollary we can always extend a trace-refinement relation corresponding to a pair of KAT expressions, to one corresponding to a pair of KAT expressions obtained from the former by enclosing them into any common context, more formally stated as follows.

Corollary 4.7. *Given any KAT expressions m, l, k_1 and k_2, and trace-refinement relation $\mathbb{T}$ such that $k_1 \leq^{\mathbb{T}} k_2$, it holds that $m \cdot k_1 \cdot l \leq^{\mathbb{T}'} m \cdot k_2 \cdot l$, where $\mathbb{T}'$ is the set $\{(m \cdot r_1 \cdot l, m \cdot r_2 \cdot l, \mathcal{A}) \mid (r_1, r_2, \mathcal{A}) \in \mathbb{T}\}$.*

Finally, we present a transitivity result by extending two trace-refinement relations at hand to single one. Let $\mathbb{T}_1$ and $\mathbb{T}_2$ be two trace-refinement relations, such that for any tuple $(o_1, p_1, \mathcal{A}_1)$ in $\mathbb{T}_1$, there is a tuple $(o_2, p_2, \mathcal{A}_2)$ in $\mathbb{T}_2$, such that $p_1 \leq o_2$. For such trace-refinement relations, we define their *transitive trace-refinement relation* to be the one containing the tuples $(o_1, p_2, \mathcal{A}_1 \cup \mathcal{A}_2)$. We denote such a trace-refinement relation by $\mathbb{T}_1 \otimes \mathbb{T}_2$.

Input: Two programs C_1, C_2 and an abstraction α.
Output: A set $O = \{(l_1^1, l_2^1, \mathcal{A}^1, \alpha^1), \ldots\}$ such that
$\textsc{Translate}(C_1, \alpha') \preceq^{\text{RefRelation}(O)} \textsc{Translate}(C_2, \alpha')$
where α' is the common abstraction of O.
Algorithm: $\textsc{Synth}(C_1,C_2,\mathcal{A}, \alpha)$ // Initially let $\mathcal{A} = \emptyset$

```
k1 := Translate(C1, α)
k2 := Translate(C2, α)
cexs = KATdiff(k1,k2,𝒜)
if no cexs return {(k1,k2,𝒜, α)}
else let R = SolveDiff(k1,k2,𝒜,cexs) in
   flatmap (λ ( r1  r2  𝒜' ).
      let (D1, D2, α') = Restrict(C1, r1, C2, r2, 𝒜 ∪ 𝒜', α) in
      Synth(D1, D2, 𝒜 ∪ 𝒜', α') ) R
```

Fig. 3. The skeleton of Synth, which synthesizes trace-refinement relations for input programs C_1, C_2.

Theorem 4.8. *For any elements k, l and m in a KAT $\mathcal{K}$, and any trace-refinement relations $\mathbb{T}_1$, $\mathbb{T}_2$, if $k \preceq^{\mathbb{T}_1} l$ and $l \preceq^{\mathbb{T}_2} m$, and $\mathbb{T}_1 \otimes \mathbb{T}_2$ is defined, then $k \preceq^{\mathbb{T}_1 \otimes \mathbb{T}_2} m$.*

5 AUTOMATION

The structure of our algorithm is given in Figure 3 and depicted in Figure 4. The input to our algorithm are programs C_1, C_2 provided, for example, in a C-like source format and parsed into ASTs. Our algorithm returns trace-refinement relations for C_1, C_2, and is parametric as to whether the relations are for equivalence versus inclusion. Technically, it returns a finite set $O = \{(l_1^1, l_2^1, \mathcal{A}^1, \alpha^1), \ldots\}$ from which the trace refinement relation RefRelation(O) can be constructed by unifying to a common abstraction $\alpha = \alpha^1 \sqcup \ldots \sqcup \alpha^n$.

Our main function Synth uses several sub-components discussed below. At the high level, it begins by using Translate, analyzing C_1 and using an iteratively constructed abstraction α to obtain the KAT expression k_1 (similar for C_2, k_2), per Section 4.1. The algorithm then checks for KAT equivalence or inclusion between k_1 and k_2 with KATdiff. If no counterexamples are found, KATdiff returns k_1 and k_2, together with the current set of hypotheses $\mathcal{A}$ as a solution. On the other hand, if KATdiff does find counterexamples, they are fed into SolveDiff, which examines them along with the KAT expressions to determine what restrictions and/or hypotheses could be employed to subdivide the search space into trace classes for which we hope further refinements can be discovered. SolveDiff returns this decision, given as a list of $(r_1, r_2, \mathcal{A})$ triples. Then Restrict is used to construct increasingly restricted versions of the input programs C_1 and C_2 and new abstractions α'. These are then are considered recursively by Synth.

5.1 Sub-Procedures

We now define and discuss the sub-procedures used by Synth. We also discuss the implementation (and limitations) of these subcomponents. As noted, the overall algorithm is parameterized by whether we are looking for solutions to equivalence ($=_{\mathcal{A}}$), or simply to inclusion ($\leq_{\mathcal{A}}$). The functionality of the sub-procedures is largely the same for the two cases.

• **Translate**(C, α)**:** This sub-procedure employs the two-step abstraction described in Section 4.1 by taking as input a program C and abstraction $\alpha = (\mathcal{K}, A_S, \alpha_S, \alpha_B)$, and returning a KAT expression k in $\mathcal{K}$ such that $\langle C, \sigma\rangle \rightsquigarrow \rho$ implies that $\alpha_B(\alpha_S(\sigma)) \cdot k \cdot \alpha_B(\alpha_S(\rho)) \neq 0$. In the *implementation*, the sub-procedure first parses the input program into an AST. From this point, one could use the approach by Kozen [1997] to model a program into a KAT expression. However we need more fine-grained detail and want accurate path expressions that exclude infeasible paths. So instead, we utilize an abstract interpreter, such as Interproc, on the possibly restricted program. Interproc

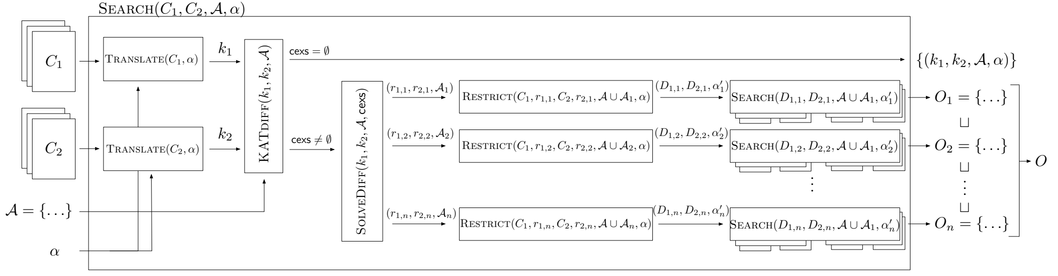


Fig. 4. An explicit depiction of the main algorithm.

populates the locations of the program with invariants according to an abstraction α[4] and annotates locations that are infeasible with $\bot$. Rather than a strictly syntactic translation, we instead exploit semantic information (*i.e.* paths of the program are determined to be infeasible) to convert the abstract program into a KAT expression k in $\mathcal{K}$ that covers its behavior. For example, consider the simple *instrumented* program `assume(d==0); c=d; if (c==0) execB() else execD();` The standard syntactic translation [Kozen 1997] alone, would produce the expression $\mathsf{d}_{\mathsf{d==0}} \cdot \mathsf{E}_{\mathsf{c=d}} \cdot (\mathsf{c}_{\mathsf{c==0}} \cdot \mathsf{B}_{\mathsf{execB}} + \overline{\mathsf{c}_{\mathsf{c==0}}} \cdot \mathsf{D}_{\mathsf{execD}})$. In our case, INTERPROC determines c==0 is always true under the instrumented `assume(d==0)` and the program is instead converted to the simpler expression $\mathsf{B}_{\mathsf{execB}}$.

• **KATDIFF**$(k_1, k_2, \mathcal{A})$**:** Given two KAT expressions k_1, k_2 and hypotheses $\mathcal{A}$, KATDIFF returns cexs, which is a set of KAT expressions k with $k \in k_1 \setminus_{\mathcal{A}} k_2$ and possibly $k \in k_2 \setminus_{\mathcal{A}} k_1$ (depending on whether we seek equivalence or inclusion). We assume this sub-procedure to be sound and complete. If cexs is empty, then the two input KAT expressions k_1 and k_2 are such that $k_1 \leq_{\mathcal{A}} k_2$ (or $k_1 =_{\mathcal{A}} k_2$). Our *implementation* uses SYMKAT [Pous 2016], which only obtains a *singleton* set cexs, and is thus either (i) a single string c in $k_1 \setminus_{\mathcal{A}} k_2$ when we seek inclusion, (ii) a single string c in $k_2 \setminus_{\mathcal{A}} k_1$ (resp. in $k_1 \setminus_{\mathcal{A}} k_2$) when we seek equivalence but $k_1 \leq_{\mathcal{A}} k_2$ (resp. $k_2 \leq_{\mathcal{A}} k_1$) holds, or (iii) a pair of strings (c_1, c_2), with c_1 in $k_1 \setminus_{\mathcal{A}} k_2$ and c_2 in $k_2 \setminus_{\mathcal{A}} k_1$, when we seek equivalence and neither $k_1 \leq_{\mathcal{A}} k_2$ or $k_2 \leq_{\mathcal{A}} k_1$ holds. If $k_1 = a \cdot M \cdot (b \cdot F + \overline{b} \cdot G)$ and $k_2 = a \cdot M \cdot \overline{b} \cdot G$, then the string $a \cdot M \cdot b \cdot F$ is included in k_1 but not in k_2, and thus in $k_1 \setminus_{\mathcal{A}} k_2$.

• **SOLVEDIFF**$(k_1, k_2, \mathcal{A}, \textbf{cexs})$**:** This procedure takes KAT expressions k_1 and k_2, a set of hypotheses, and the set of counterexamples cexs above. It returns a set R of tuples $(r_1, r_2, \mathcal{A}_r)$, each called a *restriction*. The set of restrictions R has the property that $\mathsf{proj}_1(R)$ partitions k_1, ensuring that we have completely covered all traces. Furthermore, in the interest of *progress*, we also assume that each counterexample in cexs is not a counterexample for $k_1 \cap r_1 \setminus_{\mathcal{A}_r} k_2 \cap r_2$, or even for $k_2 \cap r_2 \setminus_{\mathcal{A}_r} k_1 \cap r_1$ depending on whether equivalence is considered instead of inclusion. In our *implementation*, we apply a customized edit-distance algorithm discussed in Section 6, which returns a set of *transformations* that can be applied to two KAT strings c_1 and c_2 to make them equivalent. These transformations are in the form of removing alphabet symbols from the strings at particular locations, or replacing some symbol with another. From these transformations, SOLVEDIFF constructs a list of restrictions to be applied on the input programs of the form $(r_1, r_2, \mathcal{A}_r)$, where r_1 and r_2 are KAT expressions, and $\mathcal{A}_r$ is a set of hypotheses.

When the edit-distance algorithm asks for the removal of an alphabet symbol from, say, string c_1, we consider two cases, depending on whether the symbol corresponds to a boolean condition or not. If so, the KAT expression r_1 corresponding to this transformation is essentially obtained by adding a hypothesis inserting the valuation of the boolean variable, in the given KAT expression

[4]In our experiments, we used the convex polyhedra domain.

k_1. Since we want these restrictions to cover all behaviors of the input programs, we also consider the negation of that valuation. As such, at least two restrictions are considered, namely $(r_1, r_2, \mathcal{A}_r)$ and $(r_1', r_2', \mathcal{A}_r')$, such that r_1 and r_1' cover k_1. On the other hand, when the removal of an event M is required, a hypothesis of the form $\mathsf{M} = 1$ is added to the set of hypotheses.

• **Restrict**$(C_1, r_1, C_2, r_2, \mathcal{A}, \alpha)$ obtains new programs from previous ones, using restrictions from SolveDiff. Given programs C_1, C_2, KAT restrictions r_1, r_2, a set of hypotheses $\mathcal{A}$ and current abstraction α, this sub-procedure returns a tuple (D_1, D_2, α'), where D_1, D_2 are the new programs and α' is a new abstraction that refines α, such that, for $i \in \{1, 2\}$, $\textsc{Translate}(D_i, \alpha') \leq_{\mathcal{A}} \textsc{Translate}(C_i, \alpha) \cap r_i$, but $\textsc{Translate}(D_i, \alpha') =_{\mathcal{A}} \textsc{Translate}(C_i, \alpha') \cap r_i$. In other words, the KAT expression obtained from the new program D_i under the new refined abstraction α', is included in the KAT expression from the original program C_i under the old abstraction α using the restriction r_i, but at the same time, if we used the new abstraction α' to translate the program C_i under the restriction r_i, into a KAT expression we would obtain the same as by just translating D_i under the new abstraction. Our *implementation* restricts programs by instrumenting `assume` statements on appropriate lines of code. For example, for a program $(\mathsf{b}_{\mathsf{l=true}} \cdot \mathsf{O}_{\mathbf{log}} + \overline{\mathsf{b}_{\mathsf{l=true}}} \cdot 1)^*$ we can implement restriction $r = (\mathsf{b}_{\mathsf{l=true}} \cdot (\mathsf{b}_{\mathsf{l=true}} \cdot \mathsf{O}_{\mathbf{log}} + \overline{\mathsf{b}_{\mathsf{l=true}}} \cdot 1))^* = (\mathsf{b}_{\mathsf{l=true}} \cdot \mathsf{O}_{\mathbf{log}})^*$ with an `assume(l==true)` instrumented immediately inside the body of the corresponding while loop.

5.2 Formal Guarantees

The key challenge is soundness, even under the sub-procedure assumptions noted above, and the proof is in Appendix A.

Theorem 5.1 (Soundness). *For all C_1, C_2, and abstractions α, let $O = \textsc{Synth}(C_1, C_2, \emptyset, \alpha)$, let α' be the common abstraction of O and let $k_1 = \textsc{Translate}(C_1, \alpha')$ and $k_2 = \textsc{Translate}(C_2, \alpha')$. Then $k_1 \preceq^{RefRelation(O)} k_2$.*

Weak completeness is easier because, as per Remark 4.5, trivial solutions can be constructed. So we are more interested in generating increasingly precise solutions. For progress, as long as the sub-procedure SolveDiff returns restrictions that handle the counterexamples returned by KATdiff, then these counterexamples will not be seen again in the recursive steps that follow.

6 EDIT-DISTANCE ON EXPRESSIONS AND STRINGS

Our main algorithm depends on SolveDiff to examine a pair of KAT expressions k_1, k_2, a set of hypotheses $\mathcal{A}$, as well as counterexamples to their equivalence, and determine appropriate restrictions r_1, r_2 and additional hypotheses $\mathcal{A}'$ that could be used to further search for trace classes of k_1 that are contained in k_2, up to hypotheses $\mathcal{A} \cup \mathcal{A}'$. To achieve this, SolveDiff tries to identify the differences between the KAT expressions k_1 and k_2, or between their string-based counterexamples, and attempts to find restrictions of least impact, to apply to the two input programs. As such, we implemented a sub-procedure Distance, that takes as inputs two KAT strings c_1 and c_2, or two KAT expressions k_1 and k_2, and returns a list of *scored* transformations to be applied on the two strings (or KAT expressions) in order to make them equivalent. In our implementation we use the custom edit-distance algorithm only on counterexample strings, and in Section 6.2, we discuss how the global edit-distance for *general* KAT expressions can help in conjunction with the composition results of Section 4.3. The edit-distance on such KAT expressions has to handle the structure of the expressions, and is naturally more involved than the linear one on strings. (The former is more similar to trees [Bille 2005].) The idea behind the sub-procedure Distance is similar to edit-distance algorithms in the literature for comparing two strings/trees/graphs [Bille 2005]. These edit-distance algorithms, return a sequence of usually simple single-symbol transformations

that are classified as symbol removals, insertions, and replacements, that equate the two input strings when they are applied on them.

We had to customize edit distance for our purposes of cross-program correlation. We thought that inserting a symbol in one of the two strings or KAT expressions, is less natural than removing another one from the other string or expression, and encode such insertions in one string as removals from the other. Therefore we employ just removal and replacement transformations on the two inputs:

- Remove(c, s): Returns a new string obtained from s with the symbol c removed,
- Replace(c_1, c_2, s): Returns a new string obtained from s with the symbol c_1 replaced with the symbol c_2.

Note that each copy of each symbol in the string is uniquely labeled, and the transformations above speak about these labeled symbols, making the order in which the transformations are applied irrelevant. Moreover, in our experience, certain transformations have more impact, or are in some way *heavier* than others. As such, we attempt to score them, and use the score of each individual transformation to ultimately score the whole sequence of transformations. For example, replacing an event symbol M in some string with another symbol N, is certainly a transformation that is semantically more involved than simply setting a boolean symbol c to true. The full algorithm for edit-distance can be found below in Section 6.1.

Example 6.1. Consider the two input strings $s_1 = \mathsf{a} \cdot \mathsf{A} \cdot \mathsf{B}$ and $s_2 = \mathsf{d} \cdot \mathsf{e} \cdot \mathsf{B}$. Running the procedure Distance on s_1 and s_2, will return the pair (T, S), where T is the sequence of transformations [Replace$(\mathsf{a}, \mathsf{d}, s_1)$, Remove$(\mathsf{e}, s_2)$, Remove$(\mathsf{A}, s_1)$] and the score S = replace_scr + 2 ∗ remove_scr, where replace_scr is the cost of replacing one symbol with another of same type, and remove_scr is the cost of removing a symbol from one of the input strings. With this sequence of transformations, both strings become equal to $\mathsf{d} \cdot \mathsf{B}$.

The sequence of transformations returned by Distance is converted into the one SolveDiff returns, as follows, for A, B action symbols and a, b boolean symbols.

- Remove(A, s): add a new hypothesis A = 1
- Remove(a, s): perform case analysis and include two tuples of restrictions, resp. corresponding to setting a to true and setting a to false
- Replace(A, B, s): add a new hypothesis A = B
- Replace(a, b, s): add a new hypothesis a = b

6.1 Edit-Distance Algorithm

The algorithm, shown in Figure 5, traverses recursively the two inputs one symbol at a time, with the option of staying stationary on one of them at each iteration, and assigns a score on the association between the symbols at hand. For this, 4 different types of scores (in the form of rationals), are calcualted for any two strings, and are added to the total score at each iteration, depending on the action that is chosen. All possible cases are considered by the algorithm, and the association that leads to the smallest global score is finally chosen. The 4 different types of scores are as follows.

- remove_scr: used when a symbol is removed from one of the two strings.
- replace_scr: used when a symbol is replaced with another symbol in one of the two strings.
- match_scr: used when a symbol in one string is matched with a symbol of the same type (boolean or event) in the other string.
- penalty_scr: used when a matching such as the one above is chosen, but where the matching is between symbols of different type.

```
Input: Two strings s1, s2.
Output: A set T of transformations and a total score for that set.
Algorithm: Distance(s1, s2)
    if (s1 = [] and s2 = []) return ([], match_scr)
    else if (s1 = []) return (RemoveAll(s2), len(s2) * remove_scr)
    else if (s2 = []) return (RemoveAll(s1), len(s1) * remove_scr)
    else
      s1 = h1 ::: t1 and s2 = h2 ::: t2
      (T1, S1) =Distance(t1, s2), g1 = Remove(h1, s1), o1 = remove_scr
      (T2, S2) =Distance(s1, t2), g2 = Remove(h2, s2), o2 = remove_scr
      (T3, S3) =Distance(t1, t2)
      if (same_symbol(h1, h2))
        g3 = Match(h1, h2, s1, s2), o3 = match_scr
      else
        g3 = Replace(h1, h2, s1)
        o3 = (same_type(h1, h2)) ? replace_scr : penalty_scr
      for minimum Si:
         return (gi ::: Ti, Si + oi)
```

Fig. 5. The distance algorithm for two counterexample strings.

The values for remove_scr and replace_scr are usually 1, whereas the penalty_scr is higher than them, and correlated with the length of the input strings. On the other hand, the value of match_scr is negative, and used to counter-balance the effect of penalty_scr. In the algorithm shown above, RemoveAll(s_2), for a string $s_2 = a_1 \cdots a_n$, is shorthand for the sequence: [Remove(a_1, s_2), ..., Remove(a_n, s_2)].

6.2 Global KAT Expression Edit-Distance

We have implemented a custom edit-distance algorithm that accepts general KAT expressions as inputs, instead of merely KAT strings. The edit-distance on such KAT expressions has to handle the structure of the expressions, and is naturally more involved than the linear one on strings. (The former is more similar to trees [Bille 2005].) For example, the algorithm will attempt and match a subexpression under a star operation in one expression with a similar subexpression under a star operation in the other. In our experiments, using this distance algorithm on the whole KAT expressions, instead of the counterexamples to their equivalence or inclusion, would most of the time remove and replace many symbols. Our implementation mostly does not use this facility. However, searching for edit distance globally on the KAT expressions can be exploited in the beginning of the algorithm, in order to find natural alignments between two large programs, split them into subcomponents, apply the Synth algorithm on each pair of such subcomponents, and finally use Theorem 4.6 to combine the individual results into a solution that works over the whole programs. Our use of global KAT edit distance does not require further theoretical development and we plan to use our implementation of these ideas in future work.

7 EXPERIMENTS

In this section we experimentally validate our approach, seeking to answer a few questions:

1. Can trace refinement relations be automatically synthesized for pairs of (simple) programs?
2. Can synthesized trace refinement relations capture intuitive relationships between the program pairs, that a human might expect to hold?
3. Can synthesized trace refinement relations provide new insights into relationships between the program pairs?

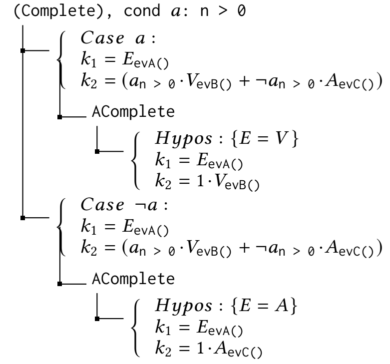
Before answering these questions, we first summarize our implementation and benchmarks.

Implementation. We have realized our algorithm in a prototype tool called Knotical. Knotical is open source, its development repository is located at https://github.com/knotical/knotical, and the artifact of this evaluation is available online [Antonopoulos et al. 2019b]. Knotical can be executed as

```
$ knotical.native -cmpLt C1 C2 0complete.c
```

to synthesize the trace refinement relations (with the argument `-cmpLt`) of the two methods `C1` and `C2` in the benchmark example `0complete.c`. To derive the trace *equivalence* relations of these methods, we can instead use the argument `-cmp`. In addition, the tool also provides the argument `-no-rem` for the users to specify the list of events/methods that cannot be removed or replaced and `-depth` to control the depth of the proof search.

Our tool is written in OCaml, using Interproc as an abstract interpreter [Lalire et al. 2009], and Symkat as a symbolic solver for KAT equalities [Pous 2015a, 2016]. We have described implementation choices made for the tool's subcomponents in Section 5.1. Knotical generates multiple solutions, internally represented in the form of trees. For the above example command-line invocation, Knotical would generate the output shown to the right. During the KATDiff and SolveDiff steps of the algorithm, multiple choices can be made and each solution tree corresponds to a particular set of choices. Branching in a solution tree corresponds to the different restrictions applied and their complement, as a result of performing case analysis on a particular condition. This can be seen in the output to the right, as the branching on the condition $n > 0$, represented via the KAT boolean test symbol a. Within each branch, Knotical identifies the restricted trace classes (denoted k_1, k_2). When hypotheses are introduced, they are denoted as *Hypos*. In this case, Knotical has generated a *complete* solution, covering all trace classes of `C1`. Often the solution trees (or subtrees) are partial, in the sense that the different restrictions applied to the programs, when taken together do not cover all behaviors of the input programs. Partial solutions can readily be converted to complete ones (as discussed in Remark 4.5).



Benchmarks. To our knowledge, no existing benchmarks were suitable for Knotical since prior works were geared towards state-based pre/post equivalence and not trace-based refinement relations. We aim to evaluate Knotical on realistic usage by considering standard systems code paradigms and some open source systems code such as `thttpd` and Merecat. Unfortunately, these systems code examples include calls to many standard C library methods and the current front-end of Knotical does not support arbitrary C language programs. As a result, we could not analyze them directly. To cope with these limitations, we manually created small benchmark C programs by extracting the key ideas and control-flow from the systems code. We treated the library methods like uninterpreted procedures and abstracted them to be KAT event symbols. Therefore our benchmarks still reflect the trace class behavior but they are yet simplified versions of the systems code.

We have evaluated our approach by applying our tool to a collection of 37 new benchmarks. Each benchmark includes two program fragments denoted C_1 and C_2. They can be found in the paper's artifact [Antonopoulos et al. 2019a,b]. Broadly speaking, our benchmarks categorized as:

- (`0*.c`) Program pairs that exercise various technical aspects of our algorithm, such as cases where refinement is trivial, where concrete refinement holds, where refinement can be achieved

Table 1. Results of applying KNOTICAL to 37 benchmarks. Those marked • are expected to have no solutions.

#	Benchmark	loc	fs	Dir	Time (s)	Sols	Tuples		Hypos	
							min	*max*	*min*	*max*
1	`0arith.c`	28	4	$=^{\mathbb{T}}$	0.03	1	1	1	2	2
2	`0complete.c`	22	5	$=^{\mathbb{T}}$	0.02	1	2	2	2	2
3	`0complete1.c`	28	6	$=^{\mathbb{T}}$	0.09	2	1	2	3	4
4	`0false.c`	15	3	$=^{\mathbb{T}}$	0.01	1	1	1	1	1
5	`0if.c`	25	5	$=^{\mathbb{T}}$	0.02	1	1	1	2	2
6	`0ifarecv.c`	27	5	$=^{\mathbb{T}}$	0.02	1	1	1	2	2
7	`0impos.c`•	19	4	$\leq^{\mathbb{T}}$	0.01	0	0	0	0	0
8	`0medstrai.c`	46	22	$=^{\mathbb{T}}$	6.18	3	1	1	2	2
9	`0needax.c`	24	4	$\leq^{\mathbb{T}}$	0.01	1	1	1	1	1
10	`0nohyp.c`	21	4	$=^{\mathbb{T}}$	0.04	2	1	1	2	2
11	`0noloop.c`	31	3	$=^{\mathbb{T}}$	0.25	2	1	1	0	1
12	`0nondet.c`	48	7	$=^{\mathbb{T}}$	0.41	2	2	2	4	4
13	`0pos.c`	22	4	$\leq^{\mathbb{T}}$	0.12	1	1	1	0	0
14	`0rename.c`	13	4	$=^{\mathbb{T}}$	0.05	1	1	1	1	1
15	`0rename1.c`	14	4	$=^{\mathbb{T}}$	0.01	1	1	1	1	1
16	`0sanity.c`	8	3	$=^{\mathbb{T}}$	0.00	1	1	1	0	0
17	`0sanity1.c`	8	3	$=^{\mathbb{T}}$	0.01	2	1	1	1	1
18	`0smstrai.c`	45	22	$=^{\mathbb{T}}$	0.66	5	1	1	1	1
19	`1acqrel.c`	38	2	$=^{\mathbb{T}}$	0.02	1	1	1	0	0
20	`1asendrecv.c`	47	8	$=^{\mathbb{T}}$	2.45	39	2	3	5	10
21	`1assume.c`	35	4	$\leq^{\mathbb{T}}$	0.98	44	1	2	3	10
22	`1concloop.c`	38	5	$=^{\mathbb{T}}$	0.72	14	1	1	1	5
23	`1concloop2.c`	34	4	$\leq^{\mathbb{T}}$	4.60	240	1	2	6	19
24	`1concloop3.c`	29	3	$=^{\mathbb{T}}$	0.69	127	1	4	3	12
25	`1linarith.c`	57	4	$=^{\mathbb{T}}$	1.20	12	1	1	4	4
26	`1loopevent.c`	36	3	$=^{\mathbb{T}}$	4.73	67	1	3	1	5
27	`1loopprint.c`	35	3	$=^{\mathbb{T}}$	0.36	12	1	2	3	5
28	`1sendrecv.c`	49	7	$\leq^{\mathbb{T}}$	3.84	75	2	7	6	19
29	`1toggle.c`	42	2	$=^{\mathbb{T}}$	0.03	1	1	1	1	1
30	`2altern.c`	25	4	$=^{\mathbb{T}}$	0.03	2	1	1	2	2
31	`2cdown.c`	23	4	$=^{\mathbb{T}}$	0.02	1	1	1	1	1
32	`2foil.c`	20	4	$=^{\mathbb{T}}$	0.01	1	1	1	1	1
33	`3buffer.c`	63	7	$=^{\mathbb{T}}$	21.07	192	1	2	6	11
34	`3syscalls.c`	59	7	$=^{\mathbb{T}}$	17.77	156	1	2	7	16
35	`4ident.c`	69	6	$=^{\mathbb{T}}$	0.50	6	1	1	3	3
36	`5thttpdEr.c`	44	10	$\leq^{\mathbb{T}}$	0.21	5	2	3	2	3
37	`5thttpdWr.c`	43	10	$\leq^{\mathbb{T}}$	1.87	62	1	2	4	8

entirely from splitting into cases, or where refinement can only be achieved through aggressive introduction of hypotheses.

- (`1*.c`) Program pairs that involve user I/O, system calls, acquire/release, and reactive web servers.
- (`2*.c`) Program pairs with tricky patterns, requiring careful alignment between two fragments.
- (`[345]*.c`) These program pairs are more challenging: `3buffer.c` and `3syscalls.c` model array access patterns with complicated array iterations, and `4ident.c` involves a larger pair with identical code.

The benchmark program pairs in `5*.c` model reactive web servers, which are the simplified versions of the `thttpd` [Poskanzer 2018] and Merecat [Nilsson 2019] HTTP servers, as described above. These two servers are related because Merecat is an extension of `thttpd` that adds SSL support and HTTP/1.1 Keep-Alive. The two programs can be decomposed into two phases: (i) writing a request, in which Merecat, unlike `thttpd`, performs compression and uses SSL to write responses

and (ii) the subsequent error handling, in which Merecat has a keep-alive option so that connections aren't closed when an error occurs. The two benchmarks `5thttpdWr.c` and `5thttpdEr.c` in Table 1 respectively correspond to these two phases and this decomposition demonstrates the compositional nature of our trace-refinement relations. Each benchmark contains a pair of code fragments taken from the `thttpd` and Merecat implementation. The code are distillations of the two servers, summarizing how they diverge in handling a request.

Evaluation. We ran Knotical on a MacBook Pro with a 3.1 GHz Intel Core i7 CPU and 16GB RAM, using the OCaml 4.06.1 compiler. Some of the generated trace-refinement relations can be found online [Antonopoulos et al. 2019a]. Table 1 summarizes these results, including the performance of Knotical. For each benchmark, we have included the lines of code (**loc**) and number of procedures (f**s**). We also indicate (**Dir**) whether the benchmark is for refinement $\leq^{\mathbb{T}}$ or for equality $=^{\mathbb{T}}$. For some of the examples, we check only $\leq^{\mathbb{T}}$ because we wanted to ensure the tool was capable of this anti-symmetric reasoning. Some of the $\leq^{\mathbb{T}}$ examples were crafted specifically for this purpose.

Next, we report the total time it took in seconds (**Time**), as well as the number of solutions discovered (**Sols**). We also report some basic statistics about the solutions generated for each benchmark. We report the number of **Tuples** in the solution that has the fewest/most (*min/max*) tuples. Similarly, we report the number of hypotheses (**Hypos**) in the solution that has the fewest/most (*min/max*) hypotheses. In general, there is no strict ordering of the weak-equivalence relations (*i.e.* which solution is better than another). We therefore rank the solutions based on heuristic orderings. For example, solutions of programs that are more similar usually involve fewer hypotheses and fewer edits. As a result, the statistics in Table 1 help show the quality of the solutions. We also evaluated the quality of the generated trace-refinement relations by carefully inspecting many of them manually and we have found that these edit-based heuristics work well.

Discussion. In most cases, Knotical was able to generate expected solutions quickly, often in fractions of a second. For simpler benchmarks (`0*.c`), there were often concise solutions with either two tuples (due to a single case-split) or one tuple (due to hypotheses). `0nondet.c` is more complicated and both of its solutions had 4 tuples. More complicated benchmarks tended to have solutions with 3 to 7 tuples. The largest number of tuples in a solution was 7 (`1sendrecv.c`) and the largest number of hypotheses in a solution was 19 (`1concloop2.c` and `1sendrecv.c`). Benchmark `1acqrel.c` had a solution with 0 hypotheses because it contains non-terminating loops, which are translated to KAT expressions 0. Benchmark `0impos.c` is expected to have no solutions because its fragments contain two different non-removable events, that cannot be made equivalent with axioms. (Knotical permits users to specify events that cannot be ignored.) Benchmarks `1concloop2.c`, `3buffer.c`, and `3syscalls.c` had hundreds of solutions because they have many complicated conditional branch and loop conditions. Case analysis on the permutations of these conditions leads to many solutions. There is not much correlation between analysis running time (or number of solutions) and lines of code. There is a stronger correlation with code complexity: many events or conditions lead to longer analysis time. For example, the analysis of `3buffer.c` took longer and yielded more solutions.

In many cases, we found that Knotical generated refinement relations that offered new insights into the relationship between program pairs. One such example is from the `05thttpdWr.c` benchmark. In this example, C_1 and C_2 correspond to the protocols of writing requests in the `thttpd` and Merecat HTTP servers, respectively. First, note that in C_2 there is a non-removable event `write_headers()` which always happens. Knotical then found a refinement relation between the trace classes of C_2 and those of C_1 (rather than vice-versa) by enforcing the nondeterministic condition `t>0`, so that `write_headers()` also always happens in C_1. Notably, there are no refinement relations for the case where `t≤0`: for any trace class in C_2, it is impossible to find a corresponding

trace class in C_1 because `write_headers()` will never be executed in C_1 in this case. In addition, in the `05thttpdEr.c` benchmark, whose C_1 and C_2 correspond to the error handling protocols in `thttpd` and Merecat, Knotical found refinement relations in which C_2's `keepalive` option is false and the procedure `clear_connection` in C_1 is removed since these features are not in the other.

We now summarize our findings, with respect to the goals stated at the beginning of this section:

(1) *Synthesis.* We found that Knotical could generate solutions for all benchmarks, except for `0impos.c` for which no solution is possible.
(2) *Intuition.* In our manual inspection of the results, we found trace refinement relations that captured our own intuition.
(3) *New insights.* We found that Knotical generated trace refinement relations that illustrate unexpected relationships between the program pairs, such as those discussed above in the `05thttpdWr.c` and `05thttpdEr.c` benchmarks.

8 RELATED WORK

Reasoning about program refinement and program equivalence is well-studied. In this section we highlight some of the most relevant approaches to the problem.

Bisimulation. A bisimilarity relation is over states and expresses that whenever one can perform an action from some state on one system, one can also perform the same action from any bisimilar state on the other system, and reach bisimilar states. Our formulation of program equivalences here (expressing program behaviors over time as KAT expressions) differs from bisimulation, which relies on step-by-step state relations. It would be challenging to enable existing notions of bisimulation to capture that $\mathsf{A} \cdot (\mathsf{B} + \mathsf{C})$ has the same behavior as $\mathsf{A} \cdot \mathsf{B} + \mathsf{A} \cdot \mathsf{C}$. It would also likely be tedious to use bisimulations to express that some events commute ($A \cdot B = B \cdot A$) or event inverses ($A \cdot B = 1$).

State-Based Semantic Differencing. In recent years a variety of works have focused on reasoning about differences between two versions of a program. Lahiri et al. [2012, 2013] describe SymDIFF, defining "differential assertion checking," which says that from an initial state that was non-failing on C, it becomes failing on C'. Their approach to assertion checking bears some similarity to self-composition [Barthe et al. 2004; Terauchi and Aiken 2005]. Partush and Yahav [2013, 2014] describe a correlating semantics based on abstract interpretation and construction of a product. Godlin and Strichman [2009] offered support for mutual recursion. Jackson and Ladd [1994] describe an approach based on input/output variable dependencies between, but do not offer formal proofs.

Others have explored how to empower relational verification by using information from relational specifications [Pick et al. 2018], and avoid repeating work on both programs. Smith et al. [2017] present a tool that learns relational specifications from input-output data. The goal of semantic differencing has also been focused on programs that interact with databases [Wang et al. 2018]. Lahiri et al. [2015] explored relational verification for approximate computing. Our notion of avoiding introducing "bad" behavior has a similar spirit to the work of Sousa et al. [2018] who focus on more complicated 3-way merges, that arise from source code merges. Others have focused on program differencing in the context of concurrent programs by considering, for example, differences in data-flow edges [Sung et al. 2018]. Wood et al. [2017] tackle program equivalence in the presence of memory allocation and garbage collection. Unno et al. [2017] describe a method of verifying relational specifications based on Horn Clause solving.

Weak Relations and Other Works. Benton [2004] introduced state-based refinement relations and provided type-theoretic and Floyd/Hoare treatments. He showed that this can be used to reason about the correctness of compiler transformations. Yang [2007] described relational reasoning for separation logic, allowing one to reason about pairs of heap-manipulating programs. As part

of the SymDiff project, Kawaguchi et al. [2010] also focused on weakening the state relations with pre/post conditions. Gyori et al. [2017] also took steps beyond concrete refinement, using equivalence relations, similar to Benton [2004] and Yang [2007], for change impact dataflow analysis.

Bouajjani et al. [2017] eschew state refinement relations in favor of a more abstract relationship between programs. They focus on concurrency questions that arise from reordering program statements and/or re-orderings due to interleaving. The authors do not work with traces in the sense defined here; rather, their traces are data-flow abstractions, represented as graphs. Person et al. [2008] described a relational symbolic execution in which they perform a state-based partitioning, based on program inputs. By contrast, our partitions arise from descriptions of the traces themselves. Trostanetski et al. [2017] focused on semantic differencing between programs, with an emphasis on scalability and modularity. They describe state-based procedure summaries that account for program differences.

There are some analogies between k-safety of a single program, and reasoning about two programs. Researchers have explored relational invariants (over multiple executions of a single program) via program transformations that "glue" copies of the program to itself, including self-composition [Barthe et al. 2004; Terauchi and Aiken 2005], product programs [Barthe et al. 2011], Cartesian Hoare logic [Sousa and Dillig 2016] and decomposition for k-safety [Antonopoulos et al. 2017]. Logozzo et al. [2014] described *verification modulo versions* and explored how necessary/sufficient environment conditions for a program C's safety can be used to determine whether program C' introduced a regression or is "correct relative to C". The work does not involve refinement relations. Barringer et al. [1984] introduced the "chop" operator and explored composition for (non-relational) temporal logic. Pous [2015b] introduced a symbolic approach for determining language equivalence between KAT expressions (see Section 5). Kumazawa and Tamai [2011] used edit distance to characterize the difference between counterexamples within a single program (infinite vs lasso traces).

9 CONCLUSION AND FUTURE WORK

We introduced *trace refinement relations*, going beyond the state refinement relations of prior works [Benton 2004; Gyori et al. 2017; Unno et al. 2017; Yang 2007]. Our relations express trace-oriented restrictions on a program behavior and case-wise correlate the behaviors of another. We have further provided a novel synthesis algorithm, based on abstract interpretation, KAT solving, restriction, and edit-distance. We have built a tool called Knotical, that synthesizes trace-refinement relations. Our experimental evaluation shows that this approach is promising.

9.1 Heaps and Arrays

The work presented in this paper is currently focused on numeric programs, without array or heap accesses. Consider, for example, a simple change made to a heap-manipulating program C_1, obtaining a new program C_2 in which the linked list is augmented so that each node contains keys and values, rather than merely values. Adopting a separation logic [O'Hearn et al. 2001] heap treatment, the linked structures could be:

$$C_1 \text{ linked list}: \quad l_1 \mapsto [v_0, n_0] * n_0 \mapsto [v_1, n_1] * n_1 \mapsto [v_2, n_2] * \cdots$$
$$C_2 \text{ linked list}: \quad l_2 \mapsto [k_0, v_0, m_0] * m_0 \mapsto [k_1, v_1, m_1] * m_1 \mapsto [k_2, v_2, m_2] * \cdots$$

In the above notation, "$*$" indicates spatial conjunction (*i.e.* that the left operand holds on one part of the heap and the right operand holds on another disjoint part of the heap), and $t \mapsto [\ldots]$ indicates a heap cell pointed to by t. C_1's nodes only have values and next pointers, while keys are added to C_2's nodes. Such a change from C_1 to C_2 may preserve some behaviors (*e.g.* enumerating all the

values) and, in other cases, new executions are possible (*e.g.* searching for a key-value pair for key k). Even when the same values are enumerated, the order of them may have changed.

An important question is whether the theory and implementation presented here can be used to support reasoning about such differences between heap-manipulating programs. This is a challenging problem because even the theoretical question of extending Kleene algebras to abstract over heaps and arrays is itself an interesting and important open problem and, even more so, in a relational setting.

Heaps and arrays are thus beyond the scope of this paper but we now remark on some considerations and possible avenues forward. A natural first step would be to consider the Relational Separation Logic by Yang [2007] which permits assertions of the form $\binom{P_1}{P_2}$ whose interpretation is that P_1 must hold of a heap h_1 while P_2 must hold of heap h_2. Yang's state relations, however, cannot express the kinds of trace behaviors we discuss in this paper. One may, on the other hand, look to extending trace-oriented logics (KAT or temporal logics) to incorporate relations over the heap. A possible step in this direction would be to extend the *evolution logic* by Yahav et al. [2006], which permits mixtures of temporal modalities and (non-relational) heap assertions. Finally, toward *inference* of (heap) trace refinement relations, it may be possible to extend the algorithms presented in this paper. However, the underlying Kleene algebra must be extended, something our framework can indeed handle in general. We refer the interested reader to the recent work by Beckett et al. [2017] for ways of extending KATs with additional theories.

9.2 Other Future Work

As discussed in Section 6.2, we plan to further explore using edit-distance at both global and local levels. Currently our framework does not yet incorporate *nested procedures* (beyond inlining), which we leave to future work. To that end, one would likely need to extend beyond KAT to Kleene Algebra with Domain [Desharnais et al. 2006]. The latter is known to enable verification of push down systems [Mathieu and Desharnais 2005].

Concurrency is also not currently supported but also appears to be feasible. First, the search space would of course be huge but one step would be to align every pair of possible program executions and eliminate inconsistent restriction relations. Second, there might be a natural fit with techniques such as counter abstraction, partial order reduction, etc. to the events/traces.

Finally, in the ongoing development of Knotical toward a more mature tool, we plan to apply it to the change history of real-world programs and evaluate it on bug fixes. To this end, we plan to model the standard C library, system calls, etc. Ultimately, we aim for Knotical to be a part of a continuous integration framework.

Another avenue is to explore how *temporal logic* properties can be adapted to trace-refinement relations. One choice would be a temporal logic such as LTL or CTL, perhaps using the LTL "chop" operator by Barringer et al. [1984] to support composition.

A OMITTED LEMMAS AND PROOFS

Given a KAT $\mathcal{K}$ and an element $k \in \mathcal{K}$, we denote the set of *guarded strings* of k by $[\![k]\!]$. For details on the definition of guarded strings, we refer the reader to the work of Kozen [2001].

Theorem A.1. *Let k and r be two KAT expressions. Then $[\![k \cap r]\!] = [\![k]\!] \cap [\![r]\!]$.*

Proof. By definition, $k \cap r$ is equal to the union of all l_i in $\mathcal{K}$, such that $l_i \leq k$ and $l_i \leq r$. Therefore, $[\![k \cap r]\!] = \bigcup_i [\![l_i]\!]$. Suppose $s \in [\![k \cap r]\!]$. Then $s \in [\![l_i]\!]$ for some i, and since $l_i \leq k$ and $l_i \leq r$ by definition, it follows that $s \in [\![k]\!]$ and $s \in [\![r]\!]$. Conversely, suppose $s \in [\![k]\!] \cap [\![r]\!]$. Then $s \leq k$ and $s \leq r$, and therefore $s = l_i$ for some i. By definition, $s \in [\![l_i]\!]$ and thus $s \in \bigcup_i [\![l_i]\!] = [\![k \cap r]\!]$. ◻

Theorem 4.3 (restated). *Let C_1 and C_2 be two programs, and let k_1 and k_2 be the two KAT expressions obtained from a strongly valid translation of the two programs respectively, under some abstraction α. Then it holds that $C_1 \preceq C_2$ if and only if $k_1 \leq k_2$.*

Proof. For what follows, we write α for $\alpha_B \circ \alpha_S$. For the *only if* direction, suppose that $C_1 \preceq C_2$ and pick any $b, d \in \mathcal{B}$. Suppose first that $b \cdot k_1 = 0$. Then let $\sigma_1, \ldots, \sigma_n$ be the set of states such that $\alpha(\sigma_1) + \ldots + \alpha(\sigma_n) = b$ for $i \leq n$. Then, for all $i \leq n$, $\langle C_1, \sigma_i \rangle \rightsquigarrow$ fault which implies that $\langle C_2, \sigma_i \rangle \rightsquigarrow$ fault by assumption that $C_1 \preceq C_2$. This means that for all $i \leq n$, $\alpha(\sigma_i) \cdot k_2 = 0$, and thus $b \cdot k_2 = \alpha(\sigma_1) + \ldots + \alpha(\sigma_n) \cdot k_2 = 0$.

The second condition states that $b \cdot k_1 \cdot d \neq 0$ implies $b \cdot k_2 \cdot d \neq 0$, or $b \cdot k_2 = 0$. Assume that $b \cdot k_1 \cdot d \neq 0$. Let $\sigma_1, \ldots, \sigma_n$ be the set of states s.t. $b = \alpha(\sigma_1) + \ldots + \alpha(\sigma_n)$, and let $\rho_1, \ldots, \rho_m$ be the set of states s.t. $d = \alpha(\rho_1) + \ldots + \alpha(\rho_m)$. Therefore, $(\alpha(\sigma_1) + \ldots + \alpha(\sigma_n)) \cdot k_1 \cdot (\alpha(\rho_1) + \alpha(\rho_m)) \neq 0$. Let (σ_i, ρ_j) be all the pairs, s.t. $\alpha(\sigma_i) \cdot k_1 \cdot \alpha(\rho_j) \neq 0$. It follows by definition that $\langle C_1, \sigma_i \rangle \rightsquigarrow \rho_j$. Therefore, either $\langle C_2, \sigma_i \rangle \rightsquigarrow \rho_j$ or $\langle C_2, \sigma_i \rangle \rightsquigarrow$fault by assumption that $C_1 \preceq C_2$. It follows that $b \cdot k_2 = (\alpha(\sigma_1) + \ldots + \alpha(\sigma_n)) \cdot k_2 = 0$ or $b \cdot k_2 \cdot d = (\alpha(\sigma_1) + \ldots + \alpha(\sigma_n)) \cdot k_2 \cdot (\alpha(\rho_1) + \ldots + \alpha(\rho_m)) \neq 0$, as required.

For the *if* direction, suppose that $k_1 \leq k_2$, and let σ, ρ be any two states in $\mathcal{S}$. If $\langle C_1, \sigma \rangle \rightsquigarrow$ fault, then $\alpha(\sigma) \cdot k_1 = 0$. By the assumption that $k_1 \leq k_2$, we have that $\alpha(\sigma) \cdot k_2 = 0$. Therefore, $\langle C_2, \sigma \rangle \rightsquigarrow$ fault. On the other hand, if $\langle C_1, \sigma \rangle \rightsquigarrow \rho$, then $\alpha(\sigma) \cdot k_1 \cdot \alpha(\rho) \neq 0$. By the assumption that $k_1 \leq k_2$, it follows that $\alpha(\sigma) \cdot k_2 \cdot \alpha(\rho) \neq 0$ or $\alpha(\sigma) \cdot k_2 = 0$. Therefore, $\langle C_2, \sigma \rangle \rightsquigarrow \rho$ or $\langle C_2, \sigma \rangle \rightsquigarrow$fault. □

Lemma A.2. *For any k, l in a KAT $\mathcal{K}$, if $k \leq l$ then $k \cap l = k$.*

Proof. By definition, $k \cap l$ is equal to $m_1 + \ldots + m_n$, where $\{m_1, \ldots, m_n\}$ is the set of all elements m in $\mathcal{K}$ such that $m \leq k$ and $m \leq l$. By assumption, k is equal to m_i for some $i \leq n$. Therefore, $k = m_1 + \ldots + m_n$ as required. □

Lemma A.3. *Suppose that $k_1, k_2 \in \mathcal{K}$ and $\mathcal{A}$ a set of hypotheses such that $k_1 \leq_{\mathcal{A}} k_2$. Then for any $\mathcal{A}'$ with $\mathcal{A} \subseteq \mathcal{A}'$, $k_1 \leq_{\mathcal{A}'} k_2$.*

Lemma A.4. *Let k_1, k_2, l_1 and l_2 be elements of a KAT $\mathcal{K}$. If $k_1 \leq l_1$ and $k_2 \leq l_2$, then $k_1 \cdot k_2 \leq l_1 \cdot l_2$ and $k_1 + k_2 \leq l_1 + l_2$.*

Proof. Firstly notice that $k_1 + l_1 = l_1$ and $k_2 + l_2 = l_2$. For the first inequality, we want to show that $k_1 \cdot k_2 + l_1 \cdot l_2 = l_1 \cdot l_2$. Using the aforementioned equalities, $k_1 \cdot k_2 + l_1 \cdot l_2 = k_1 \cdot k_2 + (k_1 + l_1) \cdot (k_2 + l_2) = k_1 \cdot k_2 + k_1 \cdot k_2 + k_1 \cdot l_2 + l_1 \cdot k_2 + l_1 \cdot l_2 = k_1 \cdot k_2 + k_1 \cdot l_2 + l_1 \cdot k_2 + l_1 \cdot l_2 = (k_1 + l_1) \cdot (k_2 + l_2) = l_1 \cdot l_2$ as required. For the second inequality, we have that $k_1 + k_2 + l_1 + l_2 = (k_1 + l_1) + (k_2 + l_2) = l_1 + l_2$ as needed. □

Lemma A.5. *Suppose that $k_1, k_2 \in \mathcal{K}$. It holds that $k_1 \leq k_2$ if and only if for all $m \in \mathcal{K}$, $m \leq k_1$ implies $m \leq k_2$.*

Proof. For the *if* direction, suppose that for all $m \in \mathcal{K}$, $m \leq k_1$ implies $m \leq k_2$. Then in particular, for $m = k_1$, $k_1 \leq k_1$ implies that $k_1 \leq k_2$.

For the *only if* direction, suppose that $k_1 \leq k_2$, and suppose for contradiction that there is $m \in \mathcal{K}$ such that $m \leq k_1$ but $m \nleq k_2$. Then $m + k_1 = k_1$, but $m + k_2 \neq k_2$. From the assumption that $k_1 \leq k_2$, it follows that $k_1 + k_2 = k_2$. Thus $k_1 + m + k_2 = k_2$, which implies that $m + k_2 = k_2$. It follows that $m \leq k_2$, which is a contradiction. □

Lemma A.6. *Let k, l, o, p be elements of some KAT $\mathcal{K}$, and let $o \leq k$ and $p \leq l$. Then $(k \cdot l) \cap (o \cdot p) \leq (k \cap o) \cdot (l \cap p)$ and $(k + l) \cap (o + p) \leq (k \cap o) + (l \cap p)$.*

Proof. We consider the first inequality first, namely, $(k \cdot l) \cap (o \cdot p) \leq (k \cap o) \cdot (l \cap p)$. By definition of intersection, we have that $(k \cdot l) \cap (o \cdot p) \leq o \cdot p$ and $(k + l) \cap (o + p) \leq o + p$. Since $o \leq k$ and $p \leq l$, by Lemma A.2, we have that $k \cap o = o$ and $l \cap p = p$. Therefore $(k \cdot l) \cap (o \cdot p) \leq (k \cap o) \cdot (l \cap p)$ and $(k + l) \cap (o + p) \leq (k \cap o) + (l \cap p)$ as required. □

Lemma A.7. *Let k, l, o, p be elements of some KAT $\mathcal{K}$. Then $(k \cap o) \cdot (l \cap p) \leq (k \cdot l) \cap (o \cdot p)$ and $(k \cap o) + (l \cap p) \leq (k + l) \cap (o + p)$.*

Proof. Consider the expressions $(k \cap o)$ and $(l \cap p)$. By definiton, $k \cap o = x_1 + \ldots + x_M$, where $\{x_1, \ldots, x_M\}$ is the set of all elements x in $\mathcal{K}$ such that $x \leq k$ and $x \leq o$. Similarly, $(l \cap p) = y_1 + \ldots + y_N$ where $\{y_1, \ldots, y_N\}$ is the set of all y in $\mathcal{K}$ such that $y \leq l$ and $y \leq p$. Therefore, by Lemma A.4, for any x in the first set and any y in the second set, $x + y \leq k + l$, $x + y \leq o + p$, $x \cdot y \leq k \cdot l$ and $x \cdot y \leq o \cdot p$.

For the first inequality, namely, $(k \cap o) \cdot (l \cap p) \leq (k \cdot l) \cap (o \cdot p)$, notice that $(k \cap o) \cdot (l \cap p)$ is equal to $(x_1 + \ldots + x_M) \cdot (y_1 + \ldots + y_M) = (x_1 \cdot y_1) + (x_1 \cdot y_2) + \ldots + (x_i \cdot y_j) + \ldots + (x_M \cdot y_M)$. Therefore, $(k \cap o) \cdot (l \cap p) \leq k \cdot l$ and $(k \cap o) \cdot (l \cap p) \leq o \cdot p$, and thus $(k \cap o) \cdot (l \cap p) \leq (k \cdot l) \cap (o \cdot p)$, as required. Similarly, for the second inequality, notice that $(k \cap o) + (l \cap p)$ is equal to $(x_1 + \ldots + x_M) + (y_1 + \ldots + y_M) = (x_1 + y_1) + (x_1 + y_2) + \ldots + (x_i + y_j) + \ldots + (x_M + y_M)$. Therefore, $(k \cap o) + (l \cap p) \leq k + l$ and $(k \cap o) + (l \cap p) \leq o + p$, and thus $(k \cap o) + (l \cap p) \leq (k + l) \cap (o + p)$. □

Lemma A.8. *Let k and o be elements of some KAT $\mathcal{K}$, and let $o \leq k$. Then $k^* \cap o^* \leq (k \cap o)^*$.*

Proof. It suffices to show that for all $n, m \in \mathbb{N}$, $k^n \cap o^m \leq (k \cap p)^*$. Let m be any element of $\mathcal{K}$, such that $m \leq k^n$ and $m \leq o^m$. Then, since $o \leq k$, by Lemma A.2 it holds that $o = k \cap o$, and therefore, $m \leq o^m$ implies that $m \leq (k \cap o)^m$, and thus $m \leq (k \cap o)^*$. Since m was chosen arbitrarily among the elements x in $\mathcal{K}$ for which $x \leq k^n \cap o^m$, by Lemma A.5, the result follows. □

Lemma A.9. *Let k and o be elements of some KAT $\mathcal{K}$. Then $(k \cap o)^* \leq k^* \cap o^*$.*

Proof. Let $M = m_1, \ldots, m_n$ be the set of all elements m, such that $m \leq k$ and $m \leq o$. Therefore, $(k \cap o)^* = (m_1 + \ldots + m_n)^*$, for $m_i \in M$. It suffices to show that for all $u \in \mathbb{N}$, $(m_1 + \ldots + m_n)^u \leq k^* \cap o^*$. In particular, it is enough to show that $u \in \mathbb{N}$, $(m_1 + \ldots + m_n)^u \leq k^u \cap o^u$. The latter is equal to $z_1 + \ldots + z_s$, for $z_i \leq k^u$ and $z_i \leq o^u$. Notice that for any element $x \leq (m_1 + \ldots + m_n)^u$, there is a function $f : [u] \to [n]$, such that $x \leq m_{f(1)} \cdot m_{f(1)} \cdots m_{f(u)}$. Since for all $j \leq u$, $m_{f(j)} \leq k$ and $m_{f(j)} \leq o$, it follows that $m_{f(1)} \cdot m_{f(1)} \cdots m_{f(u)} \leq k^u$ and $m_{f(1)} \cdot m_{f(1)} \cdots m_{f(u)} \leq o^u$. Hence, $m_{f(1)} \cdot m_{f(1)} \cdots m_{f(u)} \leq k^u \cap o^u$. Since x was chosen arbitrarily, the result follows by Lemma A.5. □

Theorem A.10. *Suppose k_1, k_2, l_1 and l_2 are KAT expressions. Let $\mathbb{T}_k$ and $\mathbb{T}_l$ be trace-refinement relations, such that $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$. Then $k_1 \cdot l_1 \leq^{\mathbb{T}_k \odot \mathbb{T}_l} k_2 \cdot l_2$.*

Proof. We want to show that for any tuple $(x, y, \mathcal{D}) \in \mathbb{T}_k \odot \mathbb{T}_l$, $(k_1 \cdot k_2) \cap x \leq_{\mathcal{D}} (l_1 \cdot l_2) \cap y$. Choose such an arbitrary tuple $(x, y, \mathcal{D}) \in \mathbb{T}_k \odot \mathbb{T}_l$, and let $(o, q, \mathcal{A}) \in \mathbb{T}_k$ and $(p, r, \mathcal{B}) \in \mathbb{T}_l$ be the tuples that produced $(x, y, \mathcal{D})$. In other words, $x = o \cdot p$, $y = q \cdot r$ and $\mathcal{D} = \mathcal{A} \cup \mathcal{B}$.

Since $\mathsf{proj}_1(\mathbb{T}_k)$ partitions k_1 and $\mathsf{proj}_1(\mathbb{T}_l)$ partitions l_1, we have that for any $o \in \mathsf{proj}_1(\mathbb{T}_k)$ and $p \in \mathsf{proj}_1(\mathbb{T}_l)$, $o \leq k$ and $p \leq l$, and thus by Lemma A.6, we have that $(k_1 \cdot l_1) \cap (o \cdot p) \leq (k_1 \cap o) \cdot (l_1 \cap p)$. By assumption, $k_1 \cap o \leq_{\mathcal{A}} k_2 \cap q$ and $l_1 \cap p \leq_{\mathcal{B}} l_2 \cap r$. Since $\mathcal{D} = \mathcal{A} \cup \mathcal{B}$, by Lemma A.3, we have that $k_1 \cap o \leq_{\mathcal{D}} k_2 \cap q$ and $l_1 \cap p \leq_{\mathcal{D}} l_2 \cap r$. Therefore, by Lemma A.4, we have that $(k_1 \cap o) \cdot (l_1 \cap p) \leq_{\mathcal{D}} (k_2 \cap q) \cdot (l_2 \cap r)$. By Lemma A.7, we have that $(k_2 \cap q) \cdot (l_2 \cap r) \leq_{\mathcal{D}} (k_2 \cdot l_2) \cap (q \cdot r)$, and hence $(k_1 \cdot l_1) \cap (o \cdot p) \leq_{\mathcal{D}} (k_2 \cdot l_2) \cap (q \cdot r)$ as required. □

Theorem A.11. *Suppose k_1, k_2, l_1 and l_2 are KAT expressions. Let $\mathbb{T}_k$ and $\mathbb{T}_l$ be trace-refinement relations, such that $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$. Then $k_1 + l_1 \leq^{\mathbb{T}_k \oplus \mathbb{T}_l} k_2 + l_2$.*

Proof. We want to show that for any tuple $(x, y, \mathcal{D}) \in \mathbb{T}_k \oplus \mathbb{T}_l$, $(k_1 + k_2) \cap x \leq_{\mathcal{D}} (l_1 + l_2) \cap y$. Choose such an arbitrary tuple $(x, y, \mathcal{D}) \in \mathbb{T}_k \oplus \mathbb{T}_l$, and let $(o, q, \mathcal{A}) \in \mathbb{T}_k$ and $(p, r, \mathcal{B}) \in \mathbb{T}_l$ be the tuples that produced $(x, y, \mathcal{D})$. In other words, $x = o + p$, $y = q + r$ and $\mathcal{D} = \mathcal{A} \cup \mathcal{B}$.

Since $\mathsf{proj}_1(\mathbb{T}_k)$ partitions k_1 and $\mathsf{proj}_1(\mathbb{T}_l)$ partitions l_1, we have that for any $o \in \mathsf{proj}_1(\mathbb{T}_k)$ and $p \in \mathsf{proj}_1(\mathbb{T}_l)$, $o \leq k$ and $p \leq l$, and thus by Lemma A.6, we have that $(k_1 + l_1) \cap (o + p) \leq (k_1 \cap o) + (l_1 \cap p)$. By assumption, $k_1 \cap o \leq_{\mathcal{A}} k_2 \cap q$ and $l_1 \cap p \leq_{\mathcal{B}} l_2 \cap r$. Since $\mathcal{D} = \mathcal{A} \cup \mathcal{B}$, by Lemma A.3, we

have that $k_1 \cap o \leq_{\mathcal{D}} k_2 \cap q$ and $l_1 \cap p \leq_{\mathcal{D}} l_2 \cap r$. Therefore, by Lemma A.4, we have that $(k_1 \cap o)+(l_1 \cap p) \leq_{\mathcal{D}} (k_2 \cap q)+(l_2 \cap r)$. By Lemma A.7, we have that $(k_2 \cap q)+(l_2 \cap r) \leq_{\mathcal{D}} (k_2+l_2) \cap (q+r)$, and hence $(k_1 + l_1) \cap (o + p) \leq_{\mathcal{D}} (k_2 + l_2) \cap (q + r)$ as required. □

Theorem A.12. *Suppose k_1, k_2, l_1 and l_2 are KAT expressions. Let $\mathbb{T}_k$ and $\mathbb{T}_l$ be trace-refinement relations, such that $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$. Then $k_1 + l_1 \leq^{\mathbb{T}_k \cup \mathbb{T}_l} k_2 + l_2$.*

Proof. Let $(x, y, \mathcal{D})$ be any tuple in $\mathbb{T}_k \cup \mathbb{T}_l$. Then $(x, y, \mathcal{D}) \in \mathbb{T}_k$ or $(x, y, \mathcal{D}) \in \mathbb{T}_l$. Since $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$, it follows by definition that $k_1 \cap x \leq_{\mathcal{D}} k_2 \cap y$ and $l_1 \cap x \leq_{\mathcal{D}} l_2 \cap y$. Notice that if either $(x, y, \mathcal{D}) \notin \mathbb{T}_k$ or $(x, y, \mathcal{D}) \notin \mathbb{T}_l$, then, respectively, either $k_1 \cap x = 0$ or $l_1 \cap x = 0$, and thus the above inequalities hold. Hence, by Lemmas A.6 and A.7, we have that $(k_1 + l_1) \cap x \leq_{\mathcal{D}} (k_2 + l_2 \cap y$ as required. □

Theorem A.13. *Given any KAT expressions k and l, and trace-refinement relation $\mathbb{T}$ such that $k \leq^{\mathbb{T}} l$, it holds that $k^* \leq^{\mathbb{T}^\star} l^*$.*

Proof. We want to show that for any tuple $(x, y, \mathcal{D}) \in \mathbb{T}^\star$, $k^* \cap x \leq_{\mathcal{D}} l^* \cap y$. Choose such an arbitrary tuple $(x, y, \mathcal{D}) \in \mathbb{T}^\star$, and let $(o, q, \mathcal{A}) \in \mathbb{T}$ be the tuple that produced $(x, y, \mathcal{D})$. In other words, $x = o^*$, $y = q^*$ and $\mathcal{D} = \mathcal{A}$.

Since $\text{proj}_1(\mathbb{T})$ partitions k_1, we have that for any $o \in \text{proj}_1(\mathbb{T})$, $o \leq k$. By Lemma A.8 and the latter inequality, it follows that $k^* \cap o^* \leq_{\mathcal{A}} (k \cap o)^*$. Then, by the assumption that $k \leq^{\mathbb{T}} l$, we have that $k \cap o \leq_{\mathcal{A}} l \cap q$, and thus $(k \cap o)^* \leq_{\mathcal{A}} (l \cap q)^*$. Furthermore, by Lemma A.9, we have that $(l \cap q)^* \leq_{\mathcal{A}} l^* \cap q^*$. Together, these inequalities give us that $k^* \cap o^* \leq_{\mathcal{A}} l^* \cap q^*$, as required. □

Theorem 4.6 (restated). *Suppose k_1, k_2, l_1 and l_2 are KAT expressions. Let $\mathbb{T}_k$ and $\mathbb{T}_l$ be trace-refinement relations, such that $k_1 \leq^{\mathbb{T}_k} k_2$ and $l_1 \leq^{\mathbb{T}_l} l_2$. Then*

- $k_1 \cdot l_1 \leq^{\mathbb{T}_k \odot \mathbb{T}_l} k_2 \cdot l_2$,
- $k_1 + l_1 \leq^{\mathbb{T}_k \oplus \mathbb{T}_l} k_2 + l_2$,
- $k_1 + l_1 \leq^{\mathbb{T}_k \cup \mathbb{T}_l} k_2 + l_2$, *and*
- $k_1^* \leq^{\mathbb{T}_k^\star} k_2^*$.

Proof. It follows immediatelly from Theorems A.10, A.11, A.12 and A.13. □

Corollary 4.7 (restated). *Given any KAT expressions m, l, k_1 and k_2, and trace-refinement relation $\mathbb{T}$ such that $k_1 \leq^{\mathbb{T}} k_2$, it holds that $m \cdot k_1 \cdot l \leq^{\mathbb{T}'} m \cdot k_2 \cdot l$, where $\mathbb{T}'$ is the set $\{(m \cdot r_1 \cdot l, m \cdot r_2 \cdot l, \mathcal{A}) \mid (r_1, r_2, \mathcal{A}) \in \mathbb{T}\}$.*

Proof. The result follows from Theorem 4.6, by noticing that $m \leq^{\mathbb{T}_m} m$ and $l \leq^{\mathbb{T}_l} l$, where $\mathbb{T}_m$ and $\mathbb{T}_l$ are the sets $\{(m, m, \emptyset)\}$ and $\{(l, l, \emptyset)\}$ respectively. □

Theorem 4.8 (restated). *For any elements k, l and m in a KAT $\mathcal{K}$, and any trace-refinement relations $\mathbb{T}_1, \mathbb{T}_2$, if $k \leq^{\mathbb{T}_1} l$ and $l \leq^{\mathbb{T}_2} m$, and $\mathbb{T}_1 \otimes \mathbb{T}_2$ is defined, then $k \leq^{\mathbb{T}_1 \otimes \mathbb{T}_2} m$.*

Proof. We want to show that for any $(o, p, \mathcal{A}) \in \mathbb{T}_1 \otimes \mathbb{T}_2$, $k \cap o \leq_{\mathcal{A}} l \cap p$. Let $(o, r_1, \mathcal{A}_1) \in \mathbb{T}_1$ and $(r_2, p, \mathcal{A}_2) \in \mathbb{T}_2$, be the two tuples that produced the tuple $(o, p, \mathcal{A})$ in their transitive trace-refinement relation. In other words, $r_1 \leq r_2$ and $\mathcal{A} = \mathcal{A}_1 \cup \mathcal{A}_2$. By assumption, we have that $k \cap o \leq_{\mathcal{A}_1} l \cap r_1$. Since $r_1 \leq r_2$, $l \cap r_1 \leq l \cap r_2$. Therefore, $k \cap o \leq_{\mathcal{A}_1} l \cap r_2$. Again by assumption, we have that $l \cap r_2 \leq_{\mathcal{A}_2} m \cap p$. By Lemma A.3, we have that $k \cap o \leq_{\mathcal{A}} l \cap r_2$ and $l \cap r_2 \leq_{\mathcal{A}} m \cap p$. Thus $k \cap o \leq_{\mathcal{A}} m \cap p$ as required. □

Theorem 5.1 (restated). (Soundness). *For all C_1, C_2, and abstractions α, let $O = \textsc{Synth}(C_1, C_2, \emptyset, \alpha)$, let α' be the common abstraction of O and let $k_1 = \textsc{Translate}(C_1, \alpha')$ and $k_2 = \textsc{Translate}(C_2, \alpha')$. Then $k_1 \leq^{RefRelation(O)} k_2$.*

Proof. Let $\mathcal{K}$ be a KAT. For a set of hypotheses $\mathcal{A}$ over $\mathcal{K}$, two KAT expressions k_1 and k_2 and a trace-refinement relation $\mathbb{T}$, we write $k_1 \preceq^{\mathbb{T}}_{\mathcal{A}} k_2$ to denote that k_1 refines k_2 with respect to $\mathbb{T}$ by augmenting the set of hypotheses with $\mathcal{A}$. We proceed by induction on the number of recursive calls to show that for any abstraction $\alpha = (\mathcal{K}, A_S, \alpha_S, \alpha_B)$, and any two programs C_1 and C_2, if $\mathbb{T} = \mathsf{RefRelation}(\textsc{Synth}(C_1, C_2, \mathcal{A}, \alpha))$, then $\textsc{Translate}(C_1, \alpha) \preceq^{\mathbb{T}}_{\mathcal{A}} \textsc{Translate}(C_1, \alpha)$. Since the algorithm is initialised with $\mathcal{A}$ being the empty set, the trace-refinement relation $\mathbb{T}$ returned will be such that $k_1 \preceq^{\mathbb{T}} k_2$.

For the base case, suppose that the algorithm returns without any recursive calls. Then, for $k_1 = \textsc{Translate}(C_1, \alpha)$ and $k_2 = \textsc{Translate}(C_2, \alpha)$, the procedure $\text{KAT}\textsc{diff}(k_1, k_2, \mathcal{A})$ returns no counterexamples. By assumption, this means that $k_1 \leq_{\mathcal{A}} k_2$, which implies that $k_1 \preceq^{\mathbb{T}}_{\mathcal{A}} k_2$, for $\mathbb{T} = \{(k_1, k_2, \mathcal{A}, \alpha)\}$.

For the inductive case, suppose that $\text{KAT}\textsc{diff}(k_1, k_2, \mathcal{A})$ returns a set of counterexamples $c = \{c_1, \dots, c_m\}$. By assumption, the subprocedure SolveDiff, given k_1, k_2, c and $\mathcal{A}$ as input, returns a set R of restrictions, say of size $n \in \mathbb{N}$, such that $\mathsf{proj}_1(R)$ partitions k_1. Let $(r_1, r_2, \mathcal{A}')$ be a tuple in R, and let (D_1, D_2, α') be the output of $\textsc{Restrict}(C_1, r_1, C_2, r_2, \mathcal{A} \cup \mathcal{A}', \alpha)$. By assumption, $\textsc{Translate}(D_1, \alpha') =_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_1, \alpha') \cap r_1$, and the same holds for D_2, C_2 and r_2. By the inductive hypothesis, if O is the output of $\textsc{Synth}(D_1, D_2, \mathcal{A} \cup \mathcal{A}', \alpha')$, then it holds that $\textsc{Translate}(D_1, \alpha') \preceq^{\mathsf{RefRelation}(O)}_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(D_2, \alpha')$.

For $i \leq n$, let $(r_{1,i}, r_{2,i}, \mathcal{A}_i)$ be the tuples in R returned by the procedure SolveDiff. For each $i \leq n$, let $(D_{1,i}, D_{2,i}, \alpha'_i)$ be the result of $\textsc{Restrict}(C_1, r_{1,i}, C_2, r_{2,i}, \mathcal{A} \cup \mathcal{A}_i, \alpha)$. Finally, let O_i be the output of $\textsc{Synth}(D_{1,i}, D_{2,i}, \mathcal{A} \cup \mathcal{A}_i)$ and $\mathbb{T}'_i$ be equal to $\mathsf{RefRelation}(O_i)$. In other words, for each $i \leq n$, let $\mathbb{T}'_i$ be the set $\{(k, l, \mathcal{A}) \mid (k, l, \mathcal{A}, \alpha'_i) \in O_i\}$, where α'_i is the common abstraction of O_i. Define β to be the abstraction $\bigsqcup_{i \leq n} \alpha'_i$, and let $\mathbb{T}_i$ be obtained from $\mathbb{T}'_i$ by having all KAT expressions be over the common abstraction β. Then define O to be equal to $O_1 \cup \dots \cup O_n$. Notice that the $\mathsf{flatmap}$ operator in the algorithm, simply returns O from all the O_i, and notice that $\mathbb{T}_1 \cup \dots \cup \mathbb{T}_n = \mathsf{RefRelation}(O_1 \cup \dots \cup O_n)$. By the argument above, we have that for all $i \leq n$,

$$\textsc{Translate}(D_{1,i}, \alpha'_i) \preceq^{\mathsf{RefRelation}(O_i)}_{\mathcal{A} \cup \mathcal{A}'_i} \textsc{Translate}(D_{2,i}, \alpha'_i), \tag{5}$$

and $\textsc{Translate}(D_{1,i}, \alpha'_i) =_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_1, \alpha') \cap r_{1,i}$. Notice that since $\mathsf{proj}_1(R)$ partitions k_1,

$$\begin{aligned}&\textsc{Translate}(C_1, \alpha') \cap r_{1,1} + \dots + \textsc{Translate}(C_1, \alpha') \cap r_{1,n}\\ &=_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_1, \alpha') \cap (r_{1,1} + \dots + r_{1,n}) =_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_1, \alpha'),\end{aligned}$$

and therefore

$$\textsc{Translate}(D_{1,1}, \alpha'_1) + \dots + \textsc{Translate}(D_{1,n}, \alpha'_n) =_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_1, \alpha'). \tag{6}$$

By a similar argument,

$$\textsc{Translate}(D_{2,1}, \alpha'_1) + \dots + \textsc{Translate}(D_{2,n}, \alpha'_n) =_{\mathcal{A} \cup \mathcal{A}'} \textsc{Translate}(C_2, \alpha'). \tag{7}$$

Therefore, by Theorem A.12 and equations (5), (6) and (7),

$$\textsc{Translate}(C_1, \alpha') \preceq^{\mathbb{T}_1 \cup \dots \cup \mathbb{T}_n}_{\mathcal{A} \cup \mathcal{A}_1 \cup \dots \mathcal{A}_n} \textsc{Translate}(C_2, \alpha'),$$

where, as was argued earlier, $\mathbb{T}_1 \cup \dots \cup \mathbb{T}_n = \mathsf{RefRelation}(O_1 \cup \dots \cup O_n) = \mathsf{RefRelation}(O)$. □

ACKNOWLEDGMENTS

The authors would also like to thank James Brotherston, David Naumann, Matthew Parkinson and the anonymous referees for their valuable comments and helpful suggestions. This work is supported by Office of Naval Research under Grant No.: N00014-17-1-2787.

REFERENCES

Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. 2017. Decomposition instead of self-composition for proving the absence of timing channels. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 362–375.

Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. 2019a. *Experimental Results of Knotical.* Retrieved August 14, 2019 from https://knotical.github.io/knotical/results/SUMMARY.html

Timos Antonopoulos, Eric Koskinen, and Ton Chanh Le. 2019b. *Knotical: An Inference System of Trace Refinement Relations.* https://doi.org/10.5281/zenodo.3368626

Howard Barringer, Ruurd Kuiper, and Amir Pnueli. 1984. Now You May Compose Temporal Logic Specifications. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, April 30 - May 2, 1984, Washington, DC, USA.* 51–63.

Gilles Barthe, Juan Manuel Crespo, and César Kunz. 2011. Relational verification using product programs. In *International Symposium on Formal Methods*. Springer, 200–214.

Gilles Barthe, Pedro R D'Argenio, and Tamara Rezk. 2004. Secure information flow by self-composition. In *CSFW*.

Ryan Beckett, Eric Campbell, and Michael Greenberg. 2017. Kleene Algebra Modulo Theories. *CoRR* abs/1707.02894 (2017). arXiv:1707.02894 http://arxiv.org/abs/1707.02894

Nick Benton. 2004. Simple relational correctness proofs for static analyses and program transformations. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2004, Venice, Italy, January 14-16, 2004.* 14–25.

Philip Bille. 2005. A survey on tree edit distance and related problems. *Theoretical computer science* 337, 1-3 (2005), 217–239.

Ahmed Bouajjani, Constantin Enea, and Shuvendu K. Lahiri. 2017. Abstract Semantic Diffing of Evolving Concurrent Programs. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings.* 46–65. https://doi.org/10.1007/978-3-319-66706-5_3

Michael R. Clarkson, Bernd Finkbeiner, Masoud Koleini, Kristopher K. Micinski, Markus N. Rabe, and César Sánchez. 2014. Temporal Logics for Hyperproperties. In *POST*. 265–284.

Jules Desharnais, Bernhard Möller, and Georg Struth. 2006. Kleene algebra with domain. *ACM Trans. Comput. Log.* 7, 4 (2006), 798–833. https://doi.org/10.1145/1183278.1183285

Benny Godlin and Ofer Strichman. 2009. Regression verification. In *Proceedings of the 46th Annual Design Automation Conference*. ACM, 466–471.

Sumit Gulwani, Sagar Jain, and Eric Koskinen. 2009. Control-flow refinement and progress invariants for bound analysis. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2009, Dublin, Ireland, June 15-21, 2009.* 375–385.

Alex Gyori, Shuvendu K. Lahiri, and Nimrod Partush. 2017. Refining interprocedural change-impact analysis using equivalence relations. In *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis, Santa Barbara, CA, USA, July 10 - 14, 2017.* 318–328. https://doi.org/10.1145/3092703.3092719

Daniel Jackson and David A Ladd. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications.. In *ICSM*, Vol. 94. 243–252.

Ming Kawaguchi, Shuvendu K Lahiri, and Henrique Rebelo. 2010. Conditional equivalence. *Microsoft, MSR-TR-2010-119, Tech. Rep* (2010).

Dexter Kozen. 1990. On kleene algebras and closed semirings. In *Mathematical Foundations of Computer Science 1990*, Branislav Rovan (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 26–47.

Dexter Kozen. 1996. Kleene Algebra withTests and Commutativity Conditions. In *Tools and Algorithms for Construction and Analysis of Systems, Second International Workshop, TACAS '96, Passau, Germany, March 27-29, 1996, Proceedings (Lecture Notes in Computer Science)*, Tiziana Margaria and Bernhard Steffen (Eds.), Vol. 1055. Springer, 14–33. https://doi.org/10.1007/3-540-61042-1_35

Dexter Kozen. 1997. Kleene Algebra with Tests. *ACM Trans. Program. Lang. Syst.* 19, 3 (1997), 427–443. https://doi.org/10.1145/256167.256195

Dexter Kozen. 2001. *Automata on Guarded Strings and Applications.* Technical Report. Ithaca, NY, USA.

Dexter Kozen. 2006. On the Representation of Kleene Algebras with Tests. In *Mathematical Foundations of Computer Science 2006, 31st International Symposium, MFCS 2006, Stará Lesná, Slovakia, August 28-September 1, 2006, Proceedings (Lecture Notes in Computer Science)*, Rastislav Kralovic and Pawel Urzyczyn (Eds.), Vol. 4162. Springer, 73–83. https://doi.org/10.1007/11821069_6

Tsutomu Kumazawa and Tetsuo Tamai. 2011. Counterexample-based error localization of behavior models. In *NASA Formal Methods Symposium*. Springer, 222–236.

Shuvendu K. Lahiri, Arvind Haran, Shaobo He, and Zvonimir Rakamaric. 2015. *Automated Differential Program Verification for Approximate Computing.* Technical Report. Microsoft Research.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs. In *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 712–717. https://doi.org/10.1007/978-3-642-31424-7_54

Shuvendu K. Lahiri, Kenneth L. McMillan, Rahul Sharma, and Chris Hawblitzel. 2013. Differential assertion checking. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 345–355. https://doi.org/10.1145/2491411.2491452

Gaël Lalire, Mathias Argoud, and Bertrand Jeannet. 2009. *Interproc analyzer for recursive programs with numerical variables*. Retrieved August 13, 2019 from http://pop-art.inrialpes.fr/people/bjeannet/bjeannet-forge/interproc/index.html

Francesco Logozzo, Shuvendu K. Lahiri, Manuel Fähndrich, and Sam Blackshear. 2014. Verification modulo versions: towards usable verification. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*. 294–304. https://doi.org/10.1145/2594291.2594326

Vincent Mathieu and Jules Desharnais. 2005. Verification of Pushdown Systems Using Omega Algebra with Domain. In *Relational Methods in Computer Science, 8th International Seminar on Relational Methods in Computer Science, 3rd International Workshop on Applications of Kleene Algebra, and Workshop of COST Action 274: TARSKI, St. Catharines, ON, Canada, February 22-26, 2005, Selected Revised Papers*. 188–199.

Laurent Mauborgne and Xavier Rival. 2005. Trace partitioning in abstract interpretation based static analyzers. In *European Symposium on Programming*. Springer, 5–20.

Carroll Morgan. 1994. *Programming from specifications*. Prentice Hall,.

Joachim Nilsson. 2019. *Merecat Embedded Web Server*. Retrieved August 13, 2019 from https://troglobit.com/projects/merecat/

Peter O'Hearn, John Reynolds, and Hongseok Yang. 2001. Local reasoning about programs that alter data structures. In *International Workshop on Computer Science Logic*. Springer, 1–19.

Peter W. O'Hearn. 2018. Continuous Reasoning: Scaling the impact of formal methods. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*. 13–25.

Nimrod Partush and Eran Yahav. 2013. Abstract Semantic Differencing for Numerical Programs. In *Static Analysis - 20th International Symposium, SAS 2013, Seattle, WA, USA, June 20-22, 2013. Proceedings*. 238–258.

Nimrod Partush and Eran Yahav. 2014. Abstract semantic differencing via speculative correlation. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*. 811–828.

Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. 226–237.

Lauren Pick, Grigory Fedyukovich, and Aarti Gupta. 2018. Exploiting Synchrony and Symmetry in Relational Verification. In *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*. 164–182.

Amir Pnueli, Michael Siegel, and Eli Singerman. 1998. Translation validation. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 151–166.

Jef Poskanzer. 2018. *thttpd HTTP server*. Retrieved August 13, 2019 from http://www.acme.com/software/thttpd/

Damien Pous. 2015a. Symbolic algorithms for language equivalence and Kleene algebra with tests. *ACM SIGPLAN Notices* 50, 1 (2015), 357–368.

Damien Pous. 2015b. Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*. 357–368. https://doi.org/10.1145/2676726.2677007

Damien Pous. 2016. *Symbolic Algorithms for Language Equivalence and Kleene Algebra with Tests*. Retrieved August 13, 2019 from https://perso.ens-lyon.fr/damien.pous/symbolickat/

Calvin Smith, Gabriel Ferns, and Aws Albarghouthi. 2017. Discovering relational specifications. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ESEC/FSE 2017, Paderborn, Germany, September 4-8, 2017*. 616–626. https://doi.org/10.1145/3106237.3106279

Marcelo Sousa and Isil Dillig. 2016. Cartesian hoare logic for verifying k-safety properties. In *ACM SIGPLAN Notices*, Vol. 51. ACM, 57–69.

Marcelo Sousa, Isil Dillig, and Shuvendu K. Lahiri. 2018. Verified three-way program merge. *PACMPL* 2, OOPSLA (2018), 165:1–165:29.

Chungha Sung, Shuvendu K. Lahiri, Constantin Enea, and Chao Wang. 2018. Datalog-based scalable semantic diffing of concurrent programs. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018*. 656–666.

Tachio Terauchi and Alex Aiken. 2005. Secure information flow as a safety problem. In *SAS*.

Anna Trostanetski, Orna Grumberg, and Daniel Kroening. 2017. Modular Demand-Driven Analysis of Semantic Difference for Program Versions. In *Static Analysis - 24th International Symposium, SAS 2017, New York, NY, USA, August 30 - September 1, 2017, Proceedings*. 405–427.

Hiroshi Unno, Sho Torii, and Hiroki Sakamoto. 2017. Automating Induction for Solving Horn Clauses. In *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II*. 571–591. https://doi.org/10.1007/978-3-319-63390-9_30

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *PACMPL* 2, POPL (2018), 56:1–56:29.

Tim Wood, Sophia Drossopoulou, Shuvendu K. Lahiri, and Susan Eisenbach. 2017. Modular Verification of Procedure Equivalence in the Presence of Memory Allocation. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*. 937–963. https://doi.org/10.1007/978-3-662-54434-1_35

Eran Yahav, Thomas W. Reps, Shmuel Sagiv, and Reinhard Wilhelm. 2006. Verifying Temporal Heap Properties Specified via Evolution Logic. *Logic Journal of the IGPL* 14, 5 (2006), 755–783. https://doi.org/10.1093/jigpal/jzl009

Hongseok Yang. 2007. Relational separation logic. *Theor. Comput. Sci.* 375, 1-3 (2007), 308–334. https://doi.org/10.1016/j.tcs.2006.12.036